MotionTheater - Bewegungsdatenanalyse in Maya

Diplomarbeit

im Studiengang Audiovisuelle Medien

an der Fachhochschule Stuttgart - Hochschule der Medien

vorgelegt von

Nico Schäfer

am 1. Juni 2004

1. Prüfer: Prof. Dr. Bernhard Eberhardt

2. Prüfer: Prof. Dr. Johannes Schaugg

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Erläuterung der Problemstellung

In dieser Diplomarbeit wurde ein Plugin für die 3D-Software Maya erstellt, mit dem es möglich ist, Motion Capture-Daten (siehe Kapitel 2.1) in Maya zu importieren und beliebig zu kombinieren. Das Plugin ermöglicht es, solche Bewegungen zu importieren, zu schneiden, zu skalieren, sie zu filtern, zwischen mehreren Bewegungen überzublenden und Bewegungen entweder auf das gesamte Skelett anzuwenden oder nur auf einzelne ausgewählte Knochen. Das Plugin wurde mit der Maya-API erstellt, die einen Zugriff auf Maya-interne Objekte erlaubt. Es wurde die Form eines Plugins gewählt, da Plugins 3-10-mal schneller ausgeführt werden als Skripte, die in der Maya-eigenen Skriptsprache MEL geschrieben sind.
Motion Capturing wird in der heutigen Zeit immer wichtiger, besonders für Computerspiele und Filme. In den letzten Jahren konnte man deutlich beobachten, dass immer mehr Produktionen aus diesen Bereichen Motion Capturing einsetzen. Der Grund dafür ist, dass manche Animationen schneller und damit billiger mittels Motion Capturing umgesetzt werden können, als die von einem professionellen Animator durch Stop-Motion oder Keyframes erstellten Animationen. Der wichtigste Grund aber ist, dass es *viel realistischere Ergebnisse* liefert. Denn ein Animator benötigt sehr viel Zeit, wenn er Bewegungen erstellen will, die wirklich menschlich sind und alle Nuancen, die dabei eine Rolle spielen, beachten

will. Das menschliche Auge erkennt sehr schnell, ob eine animierte menschliche Bewegung realistisch erscheint oder nicht. Verwendet man Motion Capturing, ist dies kein Problem, denn alle Bewegungen sind absolut realistisch, da sie ja von echten Menschen aufgenommen wurden. Allerdings gibt es hier ein Spannungsfeld zwischen Kunst (die Arbeit des Animators) auf der einen Seite und reinen Zahlen (Motion Capturing) auf der anderen. Beide Techniken besitzen ihre individuellen Vorteile: Motion Capturing bietet höchsten Realismus. Das ist allerdings nicht immer gewünscht, z.B. bei Cartoon-Charakteren. In diesem Bereich liegt die Stärke der Animatoren.
Auch an der HdM wird mittlerweile in vielen Studioproduktionen aus dem Bereich Computeranimation Maya und Motion Capturing eingesetzt. Deshalb bietet es sich an, dafür ein Tool zur Verfügung zu stellen, das eine einfache und schnelle Einbindung, Bearbeitung und Gestaltung solcher Daten ermöglicht, ohne den Umweg über dritte Softwareprogramme gehen zu müssen.

1.2 Beschreibung des Motion Capture-Datenbank-Projekts mit der Uni Bonn

Diese Arbeit stellt die Vorarbeit für ein Projekt dar, das voraussichtlich ab Oktober 2004 an der HdM in Zusammenarbeit mit der Universität Bonn durchgeführt wird. Das Ziel dieses Projekts ist die Erstellung einer Datenbank, die eine große Zahl an Motion Capture-Bewegungen enthält. Zum Aufbau dieser Datenbank wird die Motion Capture-Anlage an der HdM verwendet.
Das Institut für Informatik der Universität Bonn, das ebenfalls an dem Projekt teilnimmt, ist spezialisiert auf Datenbanken. Sie hat bereits eine Datenbank fertiggestellt, mit deren Hilfe der Anwender einen kurzen Teil eines Musikstücks singen kann (z.B. den Anfang von Beethovens fünfter Sinfonie). Die Datenbank sucht dann automatisch das angesungene Stück heraus.
Die Forschungsgruppe, die sich an der HdM an dem Projekt beteiligt, kümmert sich um die Erstellung eines Programms (ebenfalls als Maya-Plugin), mit dem der Benutzer eine Bewegung mit Hilfe einer Beschreibungssprache (z.B. einer

Skriptsprache) die Bewegung vorgeben kann, die er haben möchte. Das Programm analysiert daraufhin diese Beschreibung, sucht aus der Datenbank die Bewegungen heraus, die dem Gewünschten am besten entsprechen und erstellt dann in Echtzeit die Bewegung.

Kapitel 2

Grundlagen des Motion Capturing

2.1 Was ist Motion Capturing?

Motion Capturing ist ein Verfahren zur Aufnahme von Bewegungen menschlicher Darsteller. Die gewonnen Daten werden z.B. für im computergenerierte virtuelle Charactere verwendet, um diesen möglichst menschliche Bewegungen zu verleihen.
Es gibt verschiedene Motion Capture-Systeme: mechanische, akustische, magnetische und optische. Am verbreitetsten im Medienbereich sind die optischen Systeme. Deshalb wird deren Funktionsweise hier kurz beschrieben.
Zur Bewegungsaufnahme werden mindestens sechs Kameras verwendet, die in einem Kreis aufgestellt werden können und somit ein bestimmtes Raumvolumen definieren, in dem der Schauspieler agieren kann. Das Raumvolumen ist der Bereich, der von den Kameras erfasst wird. Der Schauspieler trägt einen schwarzen Anzug, auf dem weiße reflektierende *Marker* angebracht sind (Abbildung 2.1).
Die Kameras erfassen nun die Bewegungen, die der Darsteller ausführt. Ein Programm ermittelt, welche Marker in den Bildern, die die verschiedenen Kameras liefern, korrespondieren und berechnet daraus durch Triangulierung die tatsächliche Position jedes Markers im Raum. Die Daten werden in verschiedenen Datenformaten gespeichert und können dann von Animations-Programmen wie *Maya* von Alias|Wavefront oder *3ds max* von Discreet weiterverwendet werden.

Abbildung 2.1: Schauspieler im Anzug vor Motion Capture-Anlage

2.2 Geschichte des Motion Capturing

Die Idee, menschliche Bewegungen aufzuzeichnen, um damit Berechnungen anzustellen und Erkenntnisse zu gewinnen, gibt es schon recht lange. Bereits im späten 18. Jahrhundert analysierten Etienne Jules Marey und Edward Muybridge mit fotografischen Methoden die Bewegungen von Menschen und Tieren zu medizinischen, biologischen und militärischen Zwecken.
Edward Muybridge verwendete einen Aufbau mit 24 Kameras, um die Bewegungen eines Pferdes im Galopp zu fotografieren. Diese Kameras mussten durch Drähte ausgelöst werden, die vom aufzunehmenden Objekt (Pferd) berührt wurden.

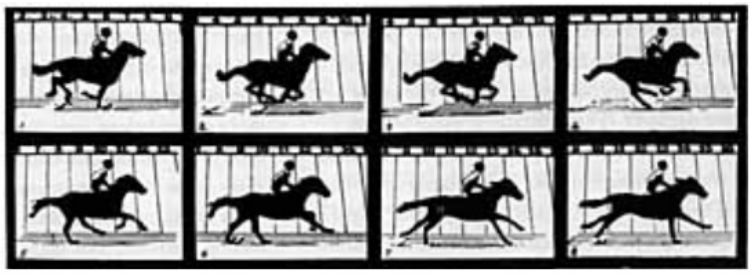


Abbildung 2.2: Edward Muybridge's Aufnahmen eines galoppierenden Pferdes [18]

Etienne Marey benutzte hingegen fotografische Langzeit- und Mehrfachbelichtung, die sogenannte *Chronophotography*, um Bewegungen aufzunehmen. Mit diesem Verfahren war es auch erstmals möglich, die Bewegungsbahnen von auf dem Körper angebrachten hellen Messpunkten festzuhalten.

Abbildung 2.3: Mareys Bewegungsanalyse [18]

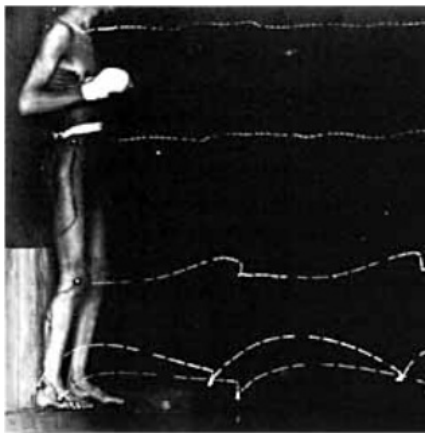


Abbildung 2.4: Bewegungsbahnen [18]

Im Jahre 1917 entwickelte der Cartoon-Zeichner Max Fleischer das Verfahren des *Rotoscoping*. Dabei wurden Bilder, die von einer Filmkamera von echten Menschen aufgenommen worden waren, auf das Zeichenbrett eines Trickfilmzeichners projiziert, die dieser dann einfach abzeichnen konnte. Diese Technik wurde u.a. 1937 von Disney für den Film *Schneewittchen* eingesetzt, um den Charakteren möglichst menschliche Bewegungen zu verleihen.

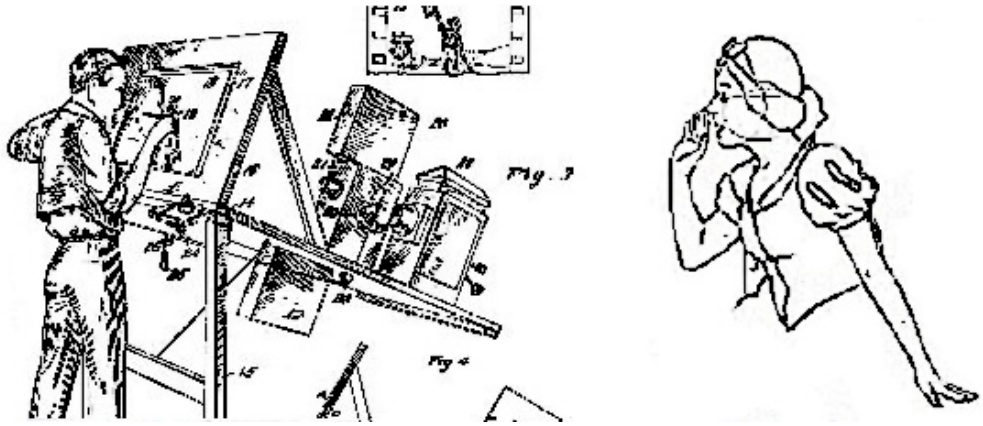


Abbildung 2.5: Max Fleischers Rotoscope & Szene aus Schneewittchen [14]

Motion Capturing in der Form, wie wir es heute kennen, hatte seinen Anfang in den späten 1970er Jahren. Damals wurden Verfahren und Geräte zur Bewegungsaufnahme für biomechanische, medizinische und militärische Zwecke entwickelt. Nach einigen Forschungsprojekten zu diesem Thema an amerikanischen Universitäten wurde die Technik schnell von Filmschaffenden aufgegriffen und auf deren Bedürfnisse angepasst. 1985 produzierte die Firma *Robert Abel and Associates* für das National Canned Food Information Council den Spot *Brilliance*. Darin wirbt ein weiblicher 3D-Roboter für Dosennahrung. Die Bewegungsdaten für diesen Roboter stammten von Trackingmarkern, die am Körper einer Schauspielerin angebracht waren. Diese wurden dann von mehreren Kameras aus verschiedenen Richtungen aufgenommen und mit Hilfe von Algorithmen ihre Position im Raum bestimmt. Anschließend wurden die so gewonnenen Bewegungsdaten als eine Art Pausvorlage verwendet, um sie so - ähnlich wie beim klassischen Rotoscoping - auf die 3D-Figur des Roboters zu übertragen.

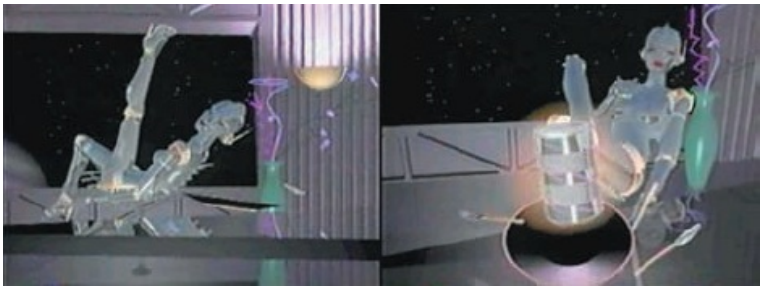


Abbildung 2.6: Weiblicher 3D-Roboter in *Brilliance* [6]

Kapitel 3

Beschreibung der Maya-API

3.1 Allgemeine Beschreibung

3.1.1 Data flow model

Maya unterscheidet sich in der Handhabung von Daten stark von anderen 3D-Programmen wie z.B. *3ds max*. Diese Programme behandeln jeden Teil der Erstellung einer 3D-Animation (Modeling, Animation, Beleuchtung, Rendering) strikt getrennt. Dies hat allerdings den Nachteil, dass solche Programme schwierig zu erweitern sind. Maya verfolgt dagegen den Ansatz, diesen Prozess auf den aller grundlegendsten Vorgang, der bei der Erstellung einer Animation verwendet wird, zu reduzieren. Der wesentliche Unterschied liegt im Begriff eines sogenannten *Knotens (Node)*, der Daten erhält, damit eine Berechnung durchführt und das Ergebnis dann an den nächsten Knoten weitergibt. Dieses Vorgehen, die Verwendung eines Netzes mit Knoten und Verbindungen zwischen ihnen, heißt *data flow model*.

3.1.2 Dependency Graph

Der *Dependency Graph* (*DG*) stellt die physikalische Repräsentation dieses Datenmodells / Netzes dar. Der DG besteht aus *Knoten* und Verbindungen zwischen diesen Knoten. Allein aus solchen Knoten und Verbindungen ist

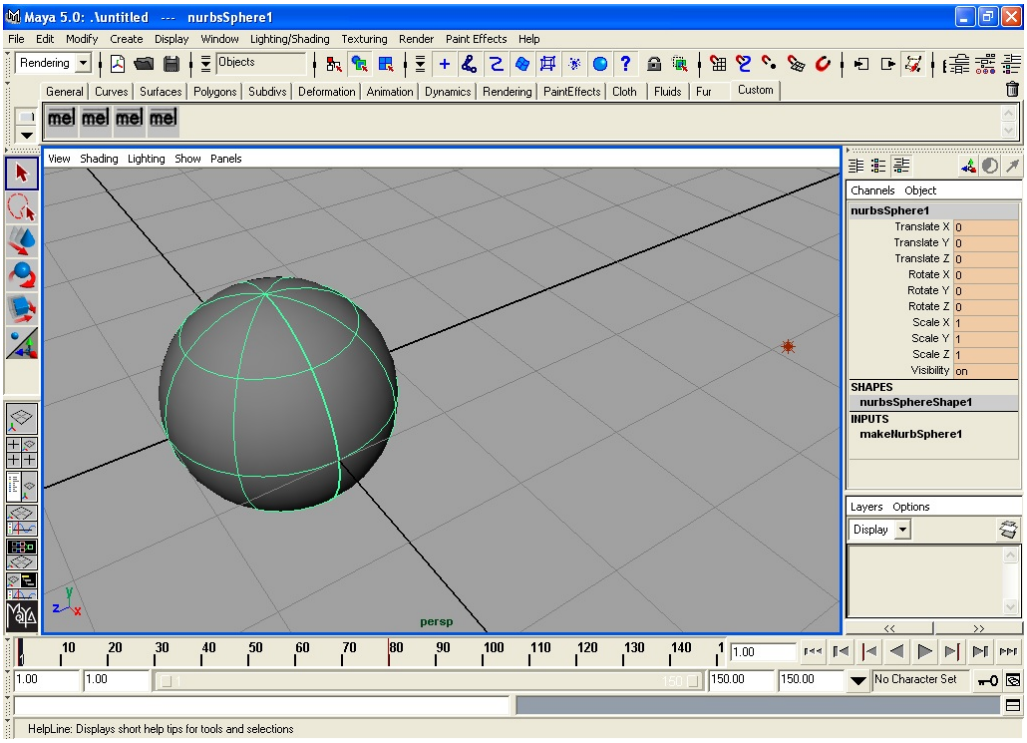


Abbildung 3.1: Einfache Szene in Maya

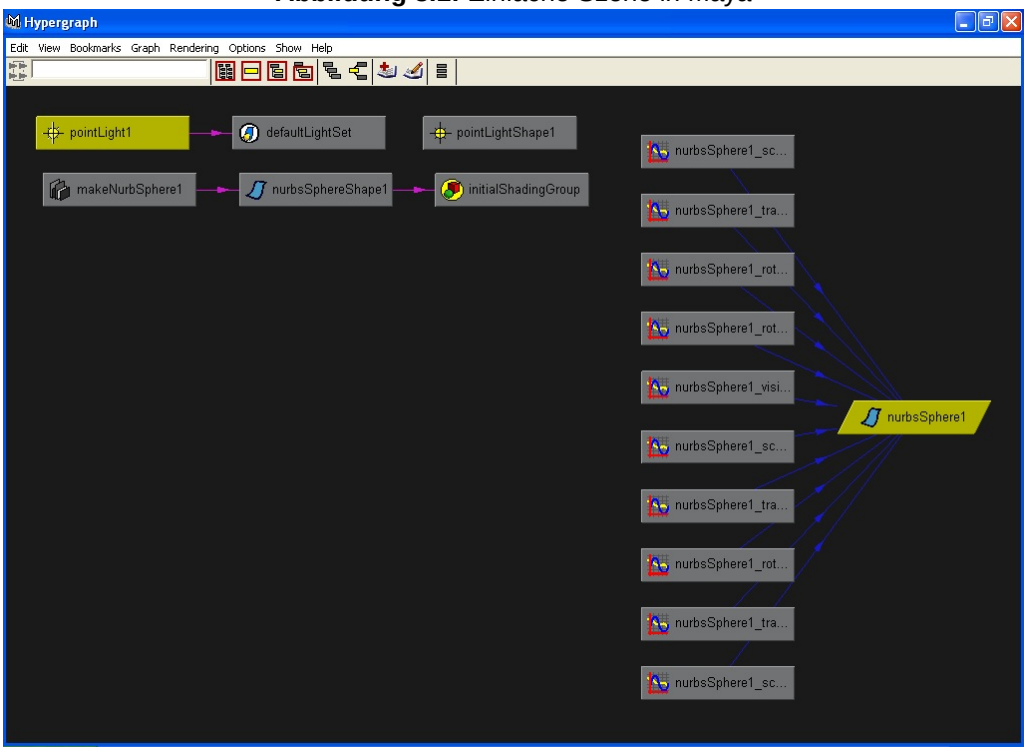


Abbildung 3.2: DG-Repräsentation der Szene

eine Maya-Szene aufgebaut. Dies beinhaltet z.B. Lichtquellen, NURBS-Objekte, Dynamics, Shading, Animationskurven usw. Abbildung 3.1 und 3.2 zeigen eine einfache Szene (oben) und Mayas DG dazu (unten).

3.1.3 Nodes, Attribute und Compute-Funktion

Jeder Knoten (Node) im DG besitzt eines oder mehrere *Attribute* und eine *compute-Funktion*. Die Attribute stellen die Eigenschaften dar, die ein Knoten besitzt. Ein Transform-Knoten, der z.B. ein Objekt in der Szene ausrichtet, besitzt u.a. ein **translateX, rotateX**-Attribut usw. Diese Attribute können etwas so Einfaches wie eine float-Zahl sein oder etwas so Komplexes wie eine ganze NURBS-Oberfläche. Die Attribute halten all diese Daten bereit, tun ansonsten aber nichts. Die compute-Funktion ist das eigentliche Herzstück jedes Knotens. Sie verarbeitet Daten aus den Attributen und führt mit ihnen eine Berechnung aus. Das Ergebnis dieser Berechnung wird wieder in einem Attribut abgelegt (in einem *Output-Attribut*), von wo es durch eine Verbindung zum nächsten Knoten weitergeleitet wird, mit dem es verbunden ist. Dieser Ansatz bietet sehr große Flexibilität und macht es für einen Programmierer einfach, neue Knoten zu definieren und sie dem Netzwerk hinzuzufügen. Dafür muss der Programmierer angeben, welche Attribute ein Knoten besitzen soll, wie der Knoten mit anderen Knoten verbunden werden soll, und er muss die compute-Funktion zur Verfügung stellen. Die API stellt dafür eine Klasse `MPxNode` bereit, von der man einfach seine eigenen Knotenklassen ableitet. Weitere wichtige Dinge, die man mit der API erstellen kann, sind MEL-Befehle (diese dienen allerdings nur zur Ansteuerung der in C++ implementierten Funktionalität), Shader, Manipulatoren, Shapes usw.

3.1.4 DAG Nodes

Eine besondere Art von DG-Nodes sind *DAG-Nodes*. Die Abkürzung DAG steht für *directed acyclic graph*. Dieser wird für alle Objekte verwendet, die in der Maya-Szene tatsächlich dargestellt werden. Im Gegensatz zum DG, der auch Dinge wie Animationskurven, Shader, Lichtquellen usw. enthalten kann, die nicht dargestellt

werden, enthält der DAG nur Objekte, die auch wirklich sichtbar sind wie NURBS- und Polygon-Objekte. Jedes dieser Objekte besteht notwendigerweise aus einem *Transform-Node* und einem *Shape-Node*. Der Transform-Node ist dem Shape-Node übergeordnet und repräsentiert Translation, Orientierung und Skalierung des Knotens. Der Shape-Node ist zuständig für die Erzeugung der Darstellung des Objekts. Der Shape-Node für eine Kugel verwendet z.B. sein *radius*-Attribut, um eine Kugelform als NURBS oder polygonales Modell in der gewünschten Größe zu erzeugen.

3.2 Konkretes Beispiel

Alle Objekte, die mit der Maya-API erzeugt werden können (Animationskurven, NURBS-Oberflächen...), haben den gleichen Typ: *MObject*. Allein anhand des Typs MObject kann man aber nicht erkennen, um welche Art von Objekt es sich tatsächlich handelt. Intern besitzt deshalb jedes MObject eine Variable, die ihren wirklichen Typ angibt, z.B. *kAnimCurve* für eine Animationskurve oder *kNurbsSurface* für eine NURBS-Oberfläche. Ein MObject besitzt allerdings nur wenige, sehr allgemeine Verwaltungs-Funktionen. Um all die verschiedenen Objekte handhaben zu können, die durch ein Objekt der Klasse MObject repräsentiert werden können, stellt die Maya-API sogenannte *Function Sets* zur Verfügung. Function Sets sind Wrapper-Klassen für die Objekte, die ein MObject repräsentieren kann. Will man z.B. eine Animationskurve in der Maya-Szene erzeugen, geht man folgendermaßen vor:

```
MFnAnimCurve fnAnimCurve = MFnAnimCurve();
MObject animCurve = MObject();
animCurve = fnAnimCurve.create(root_joint, x_rot);
```

In der ersten Zeile erzeugen wir ein Function Set für Animationskurven.
Dann erstellen wir ein MObject, dem in der letzten Zeile die Animationskurve zugewiesen wird, die *fnAnimCurve.create()* erzeugt. *root_joint* ist ein MObject vom Typ *kJoint* und gibt an, welches Attribut die Animationskurve beeinflussen soll; hier die x-Rotation des Root-Knochens (*Joints*).

Kapitel 4

Motion Theater - Programmbeschreibung

4.1 ASF/AMC-Importer

Das ASF- und AMC-File-Format zur Speicherung von Motion Capture-Daten wurde von der Firma Acclaim entwickelt. Die beiden Formate, die immer beide vorhanden sein müssen, um eine gültige Bewegung zu erhalten, sind weit verbreitet und so gut wie jedes Programm zur Aufnahme und Erzeugung von Motion Capture-Daten, z.B. der an der HdM eingesetzte *Bodybuilder* von Oxford Metrics Ltd., bieten diese Formate als Export-Option an. Aus diesem Grund verwendet *MotionTheater* ebenfalls diese Formate. Das ASF-File enthält die Definition des Skelettaufbaus, im AMC-File ist die eigentliche Bewegung gespeichert. *MotionTheater* besitzt einen Importer für ASF- und AMC-Files. Die Importer laden die angegebenen Files und erzeugen daraus in der Maya-Szene das Skelett in der Kalibrierungspose (ASF-Importer) bzw. die komplette Bewegung (AMC-Importer). Eine *Pose* ist eine stillstehende Körperhaltung des Skeletts in einem Frame. In jedem Frame befindet sich somit das Skelett in einer anderen Pose.
Es gibt zwei Verfahren zur Speicherung von Bewegungsdaten, die hauptsächlich verwendet werden. Die eine Methode speichert nur die Positionen der Motion Caputure-Marker im Raum. Ein Motion Capture-Format, das das so handhabt, ist

das CSM-Format von ???. Wir verwenden das ASF/AMC-Format, welches den zweiten Ansatz verfolgt. Dabei ist nur für die *Root* (ein Punkt, der sich genau in der Mitte des Hüftgelenks befindet) eine Position (und eine Rotation) im File vorhanden. Für alle anderen Knochen sind nur die Rotationenen im Bezug zu den ihnen hierarchisch übergeorneten Knochen gespeichert, sowie ihre Länge. Dieser Ansatz läßt sich leichter handhaben als der zuvor genannte und ist näher an dem Vorgehen, das Maya intern für die Erstellung von Knochen und Skeletten verwendet.
Im folgenden wird jeweils zuerst der Aufbau des Formats gezeigt, anschließend, wie *MotionTheater* diese Daten verarbeitet.

4.1.1 ASF-File-Format

Das ASF-File besteht aus drei Abschnitten: dem Header, der `:bonedata`-Section und der `:hierarchy`-Section.
Der Header enthält neben für *MotionTheater* unwichtigen Informationen, wie der Versionsnummer und der Dokumentation, zwei wichtige Teile: das `:units`-Tag gibt an, ob die Daten in diesem File in Grad oder Radians gespeichert sind, sowie den globalen Skalierungsfaktor für die gesamte Szene.
Wichtig ist außerdem das `:root`-Tag, welches, falls vorhanden, eine globale Translation und Rotation für die Root, also für das gesamte Skelett, enthält. Alle Knochen sind der Root hierarchisch untergeordnet. Zusätzlich ist hier die Reihenfolge gespeichert, in der die Daten im File angegeben werden. Standard ist `TX TY TZ RX RY RZ` für Translation und Rotation, manche Files speichern aber die Daten auch anders, z.B. als `TX TY TZ RZ RY RX`.
Die `:bonedata`-Section enthält Informationen darüber, aus welchen Knochen das Skelett aufgebaut ist und über die Kalibrierungspose des Skeletts. Dies ist meist eine *T-Pose* (stehende Körperhaltung mit zur Seite ausgestreckten Armen). Die Knochen werden hier einfach hintereinander (ohne Informationen über ihre Position in der Skeletthierarchie) aufgelistet. Jeder Knochen besitzt einen Namen, eine Richtung und eine Länge. Mit Hilfe der Richtung und der Länge jedes Knochens kann man die Pose, in der das Skelett steht, erzeugen. Außerdem gibt das `axis`-

Tag die Rotation des lokalen Koordinatensystems, das jeder Knochen besitzt, im Verhältnis zum lokalen Koordinatensystems des Elternknochens an. Diese vier Informationen müssen für jeden Knochen vorhanden sein.
Optional kann ein Knochen eine ID besitzen, Informationen über die Masse des Knochens und sein Massezentrum, sowie Infos über die *degrees of freedom* (`dof`-Tag). Dies sind die Freiheitsgrade = Achsen, um die das Gelenk dieses Knochens rotiert werden kann. Das `limits`-Tag gibt schließlich an, um welche Winkel das Gelenk um die eben angegebenen Achsen rotiert werden kann.
Um dies zu verdeutlichen, schauen wir uns die Definition des rechten Handgelenks aus einem ASF-File sowie Abbildung 4.1 an.

```
begin
    id 28
    name rhand
    direction -1 0 0
    length 71.4722
    axis 0 -90 -90   XYZ
    dof rx rz
    limits (-90.0 90.0)
           (-45.0 45.0)
end
```

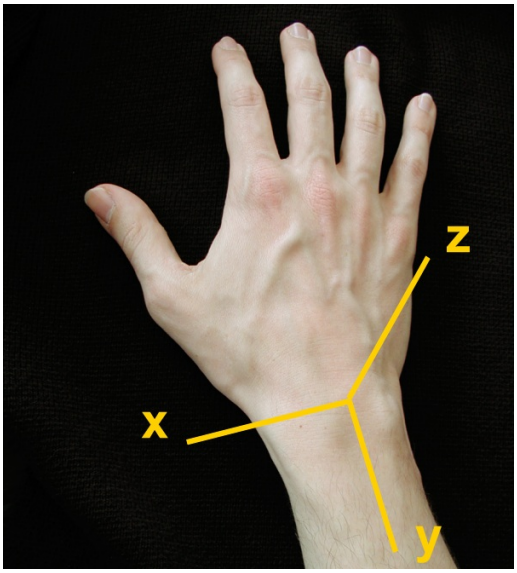


Abbildung 4.1: Drehachsen des Handgelenks

Man sieht an dem Eintrag im ASF-File, dass für die *degrees of freedom* (`dof`-Tag) nur x und z angegeben sind. Der Grund dafür ist, dass es nur möglich ist, die Hand um zwei Achsen zu drehen. Abbildung 4.1 zeigt, welches die x- und welches die z-Achse ist. Im `limits`-Tag oben kann man ablesen, dass die Hand um die x-Achse je um 90 Grad nach oben und nach unten drehbar ist, was sich leicht durch Ausprobieren bestätigen läßt. Bezüglich der z-Achse ist es nur jeweils um etwa 45 Grad möglich. Um die y-Achse läßt sich das Handgelenk nicht drehen. Eine solche Drehung kann nur ausgeführt werden, wenn man den Unterarm mit-

dreht. Deshalb sind im ASF-File für das Handgelenk x und z, aber kein y beim `dof`-Tag angegeben.

Allein aus diesen Angaben über die Knochen weiß man allerdings noch nichts darüber, wie sie miteinander verbunden sind. Die `:hierarchy`-Section gibt darüber Auskunft. Hier wird jeder Knochen, der einen Kindknochen besitzt, aufgelistet (als erster Eintrag einer Zeile), dann folgen alle Knochen, die diesen Knochen als Parent haben. Die ersten beiden Zeilen sehen z.B. so aus:

```
root lhipjoint rhipjoint lowerback
lowerback upperback
```

In der ersten Zeile wird festgelegt, dass die Root 3 Kindknochen besitzt: `lhipjoint`, `rhipjoint` und `lowerback`. Die nächste Zeile besagt, dass `lowerback` selbst Elternknochen von `upperback` ist. Die aus diesen beiden Zeilen erzeugte Konfiguration von Knochen sehen wir in Abbildung 4.2.

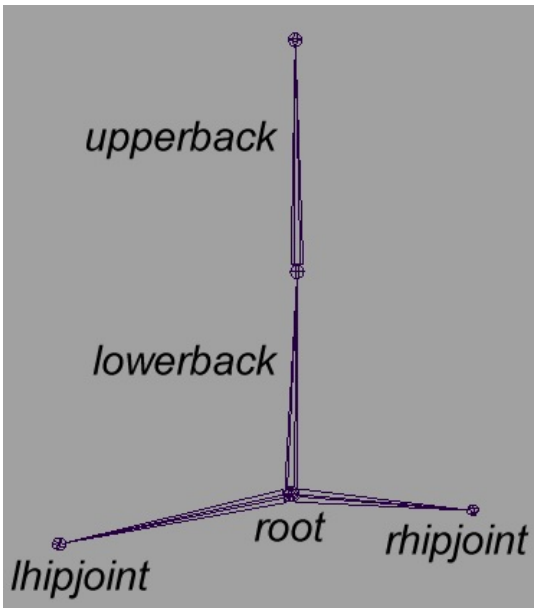


Abbildung 4.2: Knochenhierarchie ab Root-Joint

4.1.2 ASF-File-Importer

Der in *MotionTheater* integrierte ASF-File-Importer liest das vom Benutzer angegebene File ein. Zuerst werden die im Header gespeicherten Werte gelesen. Dann springt der Importer zur `:hierarchy`-Section, liest von dort die Skeletthierarchie ein und erzeugt gleichzeitig das Skelett in Maya. Nach diesem Schritt ist die Hierarchie im Maya *Hypergraph* bereits richtig vorhanden, das heißt, die Knochen sind richtig verbunden, sie liegen aber alle noch im Ursprung, da wir ja ihre Längen und Richtungen noch nicht kennen. Schließlich liest der Importer den `:bonedata`-Abschnitt des Files, um diese Informationen zu erhalten. Jeder Knochen erhält nun seine dort gespeicherte Länge und Richtung zugewiesen, sodass wir nach diesem Schritt das gesamte Skelett in der Kalibrierungspose in Maya sehen können (Abbildung 4.3).

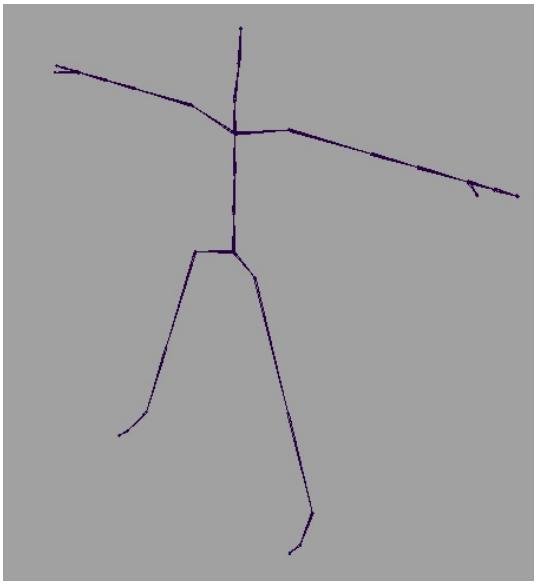


Abbildung 4.3: Skelett in Kalibrierungspose

Die Endposition jedes Knochens wird ermittelt, indem das Plugin die gesamte Hierarchie durchläuft und die Position

$$p(i) = p(i-1) + d(i) \cdot l(i) \tag{4.1}$$

für den aktuellen Knochen berechnet, wobei

$p(i)$	die Position des aktuellen Knochens,
$p(i-1)$	die Position des Elternknochens (eins höher in Hierarchie, bereits berechnet),
$d(i)$	den Richtungsvektor des aktuellen Knochens,
$l(i)$	die Länge des aktuellen Knochens

bezeichnet.
Der Importer erzeugt zusätzlich ein File namens *data_for_amc*, in dem die Informationen aus dem Header des ASF-Files und die *Degrees of freedom* für jedes Knochens für die spätere Verwendung durch den AMC-Importer abgespeichert werden. Dieser Kniff ist notwendig, da die Maya-API vorschreibt, dass jeder Befehl, den man erstellt (bei uns ist jeweils *MTImportASFFileCmd* und *MTImportAMCFileCmd* ein Befehl), nach seiner Ausführung die komplette Kontrolle an Maya zurückgeben muss. Deshalb ist es nicht möglich, Daten aus dem ASF-File, die der AMC-Importer benötigt, in Objekten oder Arrays zwischenzuspeichern. So bleibt nur der Umweg über ein File.
Um später die korrekte Bewegung zu erhalten, muss der ASF-Importer außerdem darauf achten, dass die lokalen Koordinatensysteme jedes Knochens in Maya so eingestellt werden, wie es im `axis`-Tag jedes Knochens vorgegeben ist. Betrachten wir noch einmal das Tag im oberen Codebeispiel für die rechte Hand:

```
axis 0 -90 -90   XYZ
```

Da die Spezifikation für diese Files vorgibt, dass alle Winkel lokal gespeichert werden, bedeutet diese Zeile, dass das lokale Koordinatensystem des Handgelenks im Verhältnis zur Ausrichtung des lokalen Koordinatensystems des Ellbogengelenks um jeweils -90 Grad gedreht werden muss. *MotionTheater* verwendet dazu die Funktion `MFnIkJoint.setRotateOrientation()` der Maya-API. Diese Funktion erwartet als ersten Parameter ein Quaternion, das die gewünschte Rotation des Koordinatensystems angibt. Wir müssen also die obige Zeile, die diese Werte in Eulerkoordinaten enthält, in ein Quaternion umwandeln. Wir erzeugen dafür ein eigenes Quaternion für jede negative Komponente dieses Rotationsvektors (`rotation`):

```
MVector Rx, Ry, Rz;
```

```
Rx = MVector(-rotation.x, 0, 0);
Ry = MVector(0, -rotation.y, 0);
Rz = MVector(0, 0, -rotation.z);

MQuaternion rotationQuat_x = Rx;
MQuaternion rotationQuat_y = Ry;
MQuaternion rotationQuat_z = Rz;
```

und erzeugen schließlich das eigentliche Quaternion für die Funktion `MFnIkJoint.setRotateOrientation()` durch

```
MQuaternion rotationQuat = MQuaternion();
rotationQuat = rotationQuat_z * rotationQuat_y * rotationQuat_x;
```

Man beachte, dass die Quaternionen in umgekehrter Reihenfolge multipliziert werden. Dieses Vorgehen (negativer Rotationsvektor, Multiplizieren der Achsenquaternionen in umgekehrter Reihenfolge) ist nötig, um der Funktion `setRotateOrientation()` die Rotation so zu übergeben, dass man in Maya die richtige Darstellung der Bewegung erhält.

4.1.3 AMC-File-Format

Jedes AMC-File benötigt ein zugehöriges ASF-File für die Definition des Skelettaufbaus. Ist dieses ASF-File nicht vorhanden oder nur eines, das eine andere Hierarchie enthält, wird die Bewegung nicht korrekt wiedergegeben. Das AMC-File besitzt einen kurzen Header, wovon unser Plugin nur das `:SAMPLES-PER-SECOND`-Tag ausliest und dementsprechend Maya's Wiedergabeoptionen so einstellt, dass die Bewegung mit exakt der gleichen Geschwindigkeit wiedergegeben wird, mit der sie erstellt wurde.
Der größte Teil des AMC-Files besteht aus der `:DEGREES`-Section. Diese enthält die Winkeldaten für jedes Gelenk, das im ASF-File aufgelistet ist, für alle Frames der Bewegung. Hier ein kurzer Auszug aus einem AMC-File zur Verdeutlichung:

```
1
root 192.426 854.523 474.51 14.025 -51.8006 -9.52153
lowerback -4.7254 6.58788 6.19511
upperback -0.770013 8.52742 -2.44441
thorax 3.1083 4.34633 -5.26374
...     # weitere Knochen
2
root 156.147 851.489 511.679 7.53261 -51.746 -3.83196
lowerback -2.51971 5.89277 1.33081
upperback -0.156174 7.84991 -2.26594
thorax 1.73863 3.9414 -2.71974
...
```

Die Root enthält sechs Werte: die ersten drei geben die absolute Position der Root im Raum an, die nächsten drei Werte die Orientierung/Rotation der Root. Alle anderen Knochen enthalten *nur* Rotationen, da ihre Positionen aus der Hierarchie der Knochen, ihren Richtungsvektoren und Längen ermittelt werden. Deshalb ist die Root das einzige Gelenk, für das eine konkrete Position angegeben werden muss. Die übrigen Knochen werden relativ dazu positioniert.

4.1.4 AMC-File-Importer

Der AMC-File-Importer unseres Plugins liest zuerst den Header, dann die Winkeldaten für alle Knochen in allen Frames und speichert die Werte in einem Objekt vom Typ `MTMotion` (siehe Anhang A). Anschließend erstellt er aus diesen Daten mit Hilfe der API Animationskurven. Ist dies erledigt, kann man sich schließlich die Bewegung, die im aktuellen AMC-File gespeichert ist, anschauen (Abbildung 4.4).

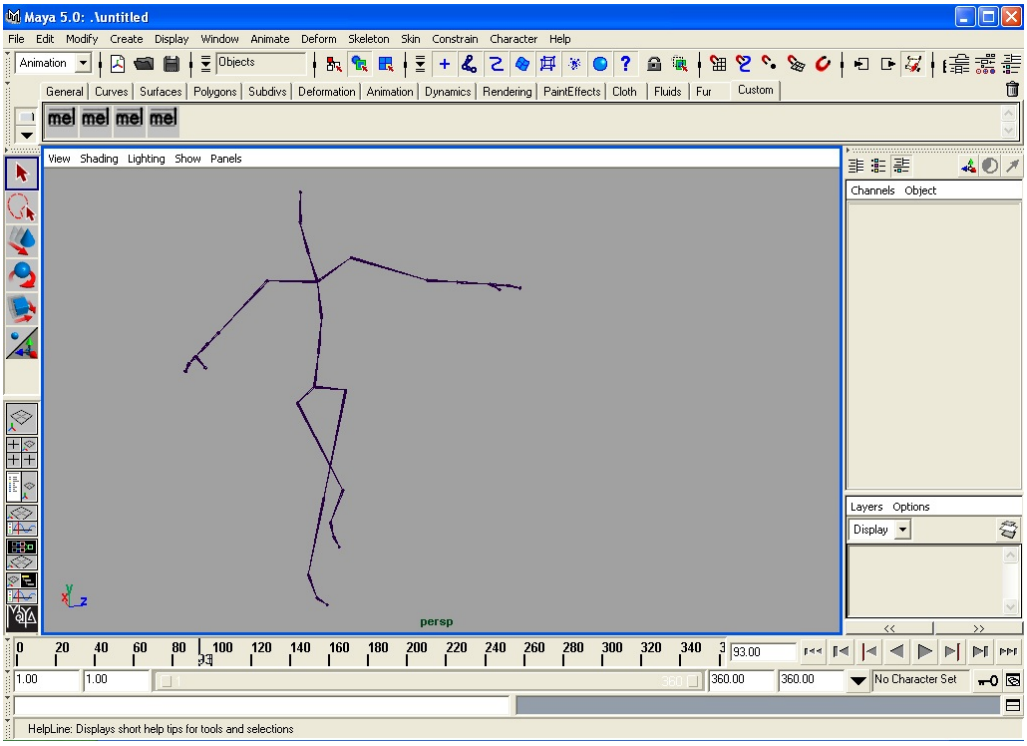


Abbildung 4.4: Die importierte Animation in Maya

Es ist in Maya möglich, Animationskurven für Positions- sowie für Rotationsdaten zu erstellen. Unser Plugin verwendet außer für die Root nur Animationskurven für Rotationsdaten, da dies einfacher zu handhaben ist. Später, bei der Weiterverarbeitung dieser Daten (Kapitel 4.2), arbeiten wir also direkt mit den im AMC-File gespeicherten Rotationen.

4.2 Motion Combiner

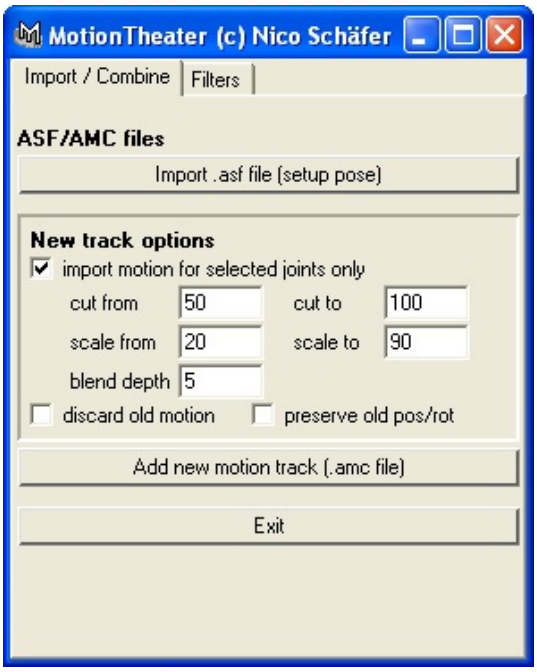


Abbildung 4.5: MotionTheater Import/Combine-Interface

Der *MotionCombiner* ist das Herzstück dieser Diplomarbeit. Mit ihm ist es möglich, mehrere Motion Capture-Files beliebig zu kombinieren. Der *MotionCombiner* besteht aus folgenden Modulen:

1. ***Cut:*** Es ist dem Anwender überlassen, ob er ganze, in einem File gespeicherte Animationen oder nur Teile davon verwenden will. Mit Hilfe der *cut from*- und *cut to*-Eingabefelder im User Interface (Abb. 4.5) des *MotionCombiners* kann er festlegen, welche Frames der Animation er importieren möchte.

2. ***Scale:*** Durch die *scale from*- und *scale to*-Eingabefelder im Interface kann man angeben, auf welche Länge die zu importierende Animation skaliert werden soll. Man kann sie so auf einen längeren oder kürzeren als den ursprünglichen Bereich skalieren und damit auch die Frames, an denen die Animation beginnt und endet, verändern.

3. ***Blend:*** Das Blend-Modul ermöglicht es, zwischen verschiedenen Bewegungen überzublenden. Dafür gibt der Anwender eine Blendtiefe (*blend depth*) an. Das Plugin ermittelt automatisch die Frames, an denen sich alte und neue Bewegungen überschneiden, denn an diesen Frames muss geblendet werden. Ist die vom Anwender angegebene Blendtiefe zu groß (d.h. sind entweder in der alten oder neuen Bewegung nicht genügend Frames vorhanden, um den Blend wie gewünscht auszuführen), ermittelt das Plugin automatisch, wie viele Frames in der alten und in der neuen Bewegungsdatei noch vorhanden sind und reduziert die Blendtiefe auf die gerade noch mögliche Anzahl an Frames. So wird dann trotzdem der maximal mögliche Blend zwischen der alten und der neuen Bewegung durchgeführt.

4. ***Preserve old position/rotation:*** Diese Funktion wird an den gleichen Grenzen zwischen alter und neuer Bewegung durchgeführt wie das Blending. Ist die *preserve old pos/rot*-Checkbox im Interface aktiviert, beginnen nachfolgende Bewegungen genau an der Position und mit der Rotation, die im letzten Frame der vorhergehenden Bewegung vorhanden waren. Das Blending und diese Option schließen sich gegenseitig aus, was automatisch vom Plugin überprüft wird. Der Anwender wird bei gleichzeitigem Anwählen beider Optionen hierüber informiert. Die Blendtiefe wird dann auf 0 gesetzt (kein Blending) und nur das Preserve-Modul wird angewandt.

5. ***Interpolate:*** Interpolation zwischen gleichen Werten in den Animationskurven ist nötig, wenn die skalierte Bewegung länger ist als die Originalbewegung. Ist dies der Fall, werden Frames aus der Originalbewegung mehrfach verwendet, um den gewünschten Bereich mit Animationswerten füllen zu können. Diese mehrfach verwendeten Frames besitzen jedoch immer den gleichen Wert, da man dafür ja keine zusätzliche Information besitzt, die man benutzen könnte. Das Interpolate-Modul ermittelt nun, wo mehrere Frames mit gleichem Wert verwendet werden und interpoliert zwischen diesen Werten und dem angrenzenden (verschiedenen) Start- und Endwert linear, um weichere Animationen zu erzeugen.

Im Laufe der Diplomarbeit gab es den *MotionCombiner* in zwei Versionen. In die erste Version wurden nach und nach die verschiedenen eben genannten Features integriert. Dabei stellte sich allerdings heraus, dass das Konzept, wie diese Dinge bis dahin umgesetzt wurden, immer komplizierter wurde.
Die zweite Version stellt ein Redesign und eine große Vereinfachung bei gleichzeitiger Erweiterung der Funktionalität dar.
Alle Module können entweder auf das gesamte Skelett angewandt werden oder nur auf einzelne Knochen, die zuvor im Maya *Hypergraph* ausgewählt wurden.

4.2.1 Alte Version: Fallunterscheidung

Die erste Version bestand hauptsächlich aus Fallunterscheidungen für alle möglichen cut-, scale- und blend-Fälle. Dies wurde allerdings sehr schnell unübersichtlich, da es allein beim cut/scale-Modul bereits 4 normale Fälle + 7 Spezialfälle gab. Beim Blend-Modul kamen nochmal 6 Fälle dazu. Das ergibt bereits (4+7)*6 = 66 Fälle, die alle zu berücksichtigen waren. Außerdem mußte darauf geachtet werden, ob die alte Bewegung erhalten oder weggeworfen werden sollte, ob die aktuelle Aktion nur für ausgewählte oder für alle Knochen durchgeführt werden sollte usw. Durch einfaches Fallunterscheiden ist dies nicht umzusetzen.

4.2.2 Neue Version: *MotionCombiner*-System

Aufgrund der eben genannten Probleme wurde es schließlich Zeit für ein Redesign. So entstand das *MotionCombiner*-System. Dieses System vereinfacht und erweitert die alte Version und seine genaue Funktionsweise wird hier nun ausführlich beschrieben.

4.2.2.1 Spezifikation

Das neue System sollte folgende Vorgaben erfüllen:

1. Keine Fallunterscheidungen mehr bzw. so wenige wie nur irgend möglich (es werden nur noch zwei Fälle unterschieden, nämlich Aktionen am Anfang

und am Ende der neu zu erstellenden Bewegung)

2. Pro Knochen nur ein Durchlauf durch die gesamte zu erstellende Bewegung, um die neuen Animationskurven zu erstellen
3. Frame-weise Bearbeitung der Animationskurven
4. Pro Frame *eine* Formel, die die komplette neue Position/Rotation berechnet. Diese Formel soll aus einzelnen Summanden bestehen, die jeweils durch einzelne Funktionen berechnet werden.
5. Es sollen einfache Hilfsfunktionen erstellt werden, die die Aktionen, welche der *MotionCombiner* durchführt, wesentlich vereinfachen. Diese Funktionen werden in Abschnitt 4.2.2.3 kurz vorgestellt.

4.2.2.2 Verwendete Bezeichnungen & Variablen

In diesem ganzen Kapitel tauchen einige Bezeichnungen/Variablen immer wieder auf, weshalb sie hier kurz erklärt werden (siehe Anhang B für eine grafische Darstellung). Der Variablenname befindet sich jeweils in der Klammer:

- m_{old} ist die alte Bewegung, die bereits in einem früheren Schritt in Maya importiert wurde.
- m_{orig} ist die neue oder Originalbewegung, die in einer Datei vorliegt und im aktuellen Schritt importiert werden soll.

  m_{cs} ist die neue geschnittene und skalierte Bewegung. *cs* steht für *cutscaled*.
- *frame_num* (i) ist die Nummer des aktuellen Frames.
- *start_frame* (f_s) *start_frame* steht für das erste Frame der neuen Bewegung. Dieses Frame wird in dieser Arbeit auch als *Start-Frame* bezeichnet. Die Bezeichnung f_s steht für $frame_{start}$, das f steht also für *frame* und die ganze Variable für *start_frame*. Die Bezeichnung wurde so gewählt, da wir auch noch Variablen für *scale_from* und *scale_to* benötigen, die mit s_f und s_t bezeichnet werden (siehe unten). Diese Variable (f_s) ist identisch mit *file_start_frame* ($f_{s,file}$).

- *end_frame* (f_e) bezeichnet das letzte Frame der neuen Bewegung; diese Variable ist identisch mit *file_end_frame* ($f_{e,file}$).
- *cut_from* (c_f) / *cut_to* (c_t) sind die vom Benutzer angegebenen Framenummern, an denen die neue Bewegung geschnitten werden soll.
- *scale_from* (s_f) / *scale_to* (s_t) sind die vom Benutzer angegebenen Framenummern, auf die die neue Bewegung skaliert werden soll.
- *blend_depth* (b) ist die vom Benutzer angegebene Blendtiefe.
- *real_start_frame* ($f_{s,real}$) / *real_end_frame* ($f_{e,real}$) geben die wirklichen Start- und End-Frames der neuen Bewegung an. Sie sind abhängig von c_f, c_t und von s_f, s_t und werden folgendermaßen ermittelt:

```
real_start_frame = motion_orig->file_start_frame;
if(cut_from!=0) real_start_frame = cut_from;
if(scale_from!=0) start_frame = scale_from;
```

Abbildung 4.6 veranschaulicht, was in diesem Codeabschnitt geschieht.

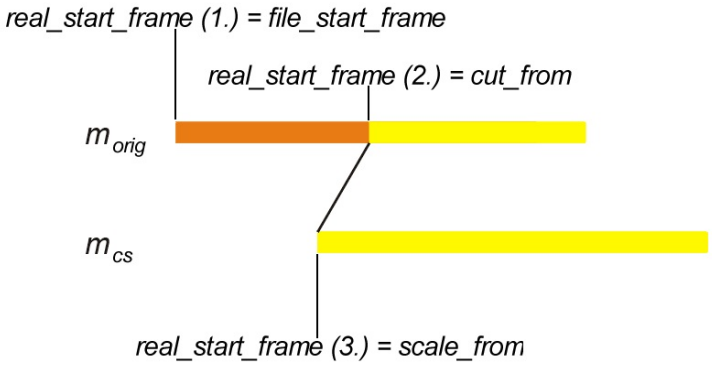


Abbildung 4.6: Grafische Darstellung des obigen Codes

motion_orig->file_start_frame enthält die Nummer des ersten Frames aus der importierten Bewegungsdatei (ist also identisch mit $f_{s,file}$). Ist *cut_from* gesetzt, wird *real_start_frame* auf *cut_from* gesetzt, da die neue Bewegung nicht bei der Framenummer anfängt, die als erstes im File gespeichert ist, sondern bei dem Frame, an dem der Anwender den Anfang schneiden will. Das gleiche passiert mit

scale_from. Der obige Code ist entsprechend mit *cut_to* und *scale_to* genauso für *real_end_frame* vorhanden.

4.2.2.3 Hilfsfunktionen

Die drei hier vorgestellten Hilfsfunktionen sind kurz, aber dennoch von großer Bedeutung für den *MotionCombiner*.

Funktion *getIndex_real()*

```
unsigned getIndex_real(unsigned frame_num)
{
    if((frame_num < real_start_frame) || (frame_num > real_end_frame))
        return 0;   // error
    else
        return (frame_num - real_start_frame + 1);
}
```

Diese Funktion ermittelt den wirklichen Index des Frames mit der gewünschten Framenummer (*frame_num*) im Array *MTMotion*. *MTMotion* ist ein Array, das alle Frames der neuen Bewegung enthält. Die Framenummer und der 1-basierte Index in diesem Array können sich unterscheiden, wie die folgende Abbildung 4.7 zeigt:

Abbildung 4.7: Beziehung zwischen Framenummer und Array-Index

getIndex_real() ermittelt diesen Array-Index.

Funktion *frameExists_file()*

```
bool MTMotion::frameExists_file(unsigned frame_num)
{
    if((frame_num >= file_start_frame) && (frame_num <= file_end_frame))
        return true;
    else return false;
}
```

Diese Funktion stellt fest, ob das Frame mit der gegebenen Framenummer (*frame_num*) im File, das die zu erstellende Bewegung enthält, existiert. Dies ist der Fall, wenn *frame_num* zwischen *file_start_frame* und *file_end_frame*, also dem ersten und dem letzten in der Datei gespeicherten Frame, liegt.

Funktion *importAMCFile()*

```
MStatus importAMCFile(const MString filename, unsigned
start_frame, unsigned end_frame, MTMotion &motion_array)
{
    // Code zu lang, um ihn hier darzustellen
    // siehe Funktionsbeschreibung
}
```

Diese Funktion lädt aus der gegebenen AMC-Datei (*filename*) alle Frames zwischen *start_frame* und *end_frame* in das der Funktion als Paramter übergebene *motion_array* (Array, das nach Ausführung der Funktion die ganze soeben importierte Bewegung enthält). Die Funktion wird verwendet, um Frames aus der alten und der neuen Bewegung nach Bedarf zu laden. Die Funktion ist außerdem sehr gut geeignet für das Mocap-Datenbank-Projekt mit der Uni Bonn, weil dafür Bewegungsdaten aus vielen verschiedenen Dateien schnell und einfach zu laden sein müssen.

4.2.2.4 Cut/Scale-Modul

Das Cut/Scale-Modul ermöglicht es dem Anwender, bei der neu zu importierenden Bewegung anzugeben, welche Frames der Bewegung er ausschneiden will und auf welchen Bereich er sie skalieren will. In Abbildung 4.8 wird das anschaulich dargestellt.

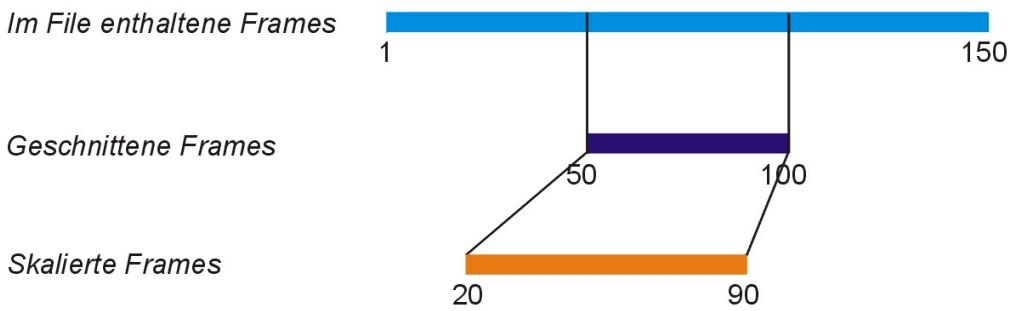


Abbildung 4.8: File –> Cut –> Scale

Es gibt hierbei vier Fälle zu berücksichtigen, die auftreten können, je nachdem, welche Werte der Anwender über das User Interface angegeben hat. Bei all diesen Fällen geht es darum, aus der Originalbewegung m_{orig} - dies ist die komplette Bewegung, die im aktuellen File enthalten ist - die geschnittene und/oder skalierte Bewegung m_{cs} (cs steht für *cutscaled*) zu ermitteln. Wir müssen also m_{cs} so erstellen, dass es die geschnittene/skalierte Version von m_{orig} ist.

Ist die skalierte Bewegung kleiner als die Originalbewegung, enthält sie weniger Frames und es gilt

$$f_{e,orig} - f_{s,orig} > f_{e,cs} - f_{s,cs} \tag{4.2}$$

In diesem Fall müssen Frames aus der Originalbewegung übersprungen werden.

Ist die skalierte Bewegung größer, enthält sie also mehr Frames als die Originalbewegung, gilt

$$f_{e,orig} - f_{s,orig} < f_{e,cs} - f_{s,cs} \tag{4.3}$$

und es müssen Frames aus der Originalbewegung mehrfach hintereinander gesetzt werden, damit man für m_{cs} genügend Frames zur Verfügung hat. Um dies

zu erreichen, durchläuft ein Parameter i (s. Abb. 4.8) einmal den Bereich, der die skalierten Frames enthalten soll, also $f_s <= i <= f_t$ oder $c_f <= i <= c_t$ oder $s_f <= i <= s_t$, je nachdem, welcher Fall vorliegt (siehe unten für die Beschreibung dieser Fälle). Wir suchen eine Formel, die i als Parameter erhält und uns das entsprechende Frame aus m_{orig} liefert, sodass, nachdem i einmal den gesamten Bereich durchlaufen hat, m_{orig} komplett auf m_{cs} umskaliert wurde.
Im folgenden werden die vier möglichen Fälle und ihre Handhabung beschrieben:

Fall 1: keine Einstellungen Wird nichts angegeben, sind die Variablen c_f, c_t, s_f und s_t 0. In diesem Fall importiert das Plugin die Daten aus den Files, die nacheinander spezifiziert werden, ohne sie zu schneiden oder zu skalieren. Abb. 4.8 zeigt zwei Bewegungen, die importiert werden. Die obere Bewegung wurde bereits in einem früheren Schritt importiert. Die mittlere Bewegung ist diejenige, die momentan importiert wird und die untere Linie zeigt die resultierende Bewegung. Die neue Bewegung wurde in diesem Fall einfach in die alte eingesetzt.

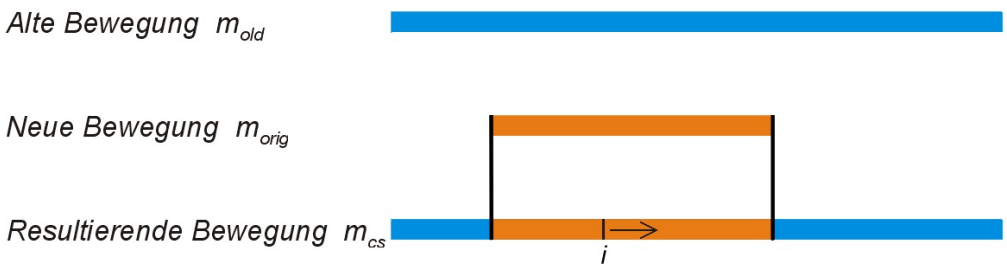


Abbildung 4.9: Resultierende Bewegung mit c_f, c_t, s_f, s_t = 0

Die Formel für i, die für jedes Frame aus m_{cs} das korrekte Frame x aus m_{orig} ermittelt, ist hier sehr einfach:

$$x = m_{orig}(i) = i \tag{4.4}$$

Da weder geschnitten noch skaliert wird, ist das gesuchte Frame aus m_{orig} identisch mit dem aktuellen Frame von m_{cs}.

Fall 2: nur Cut

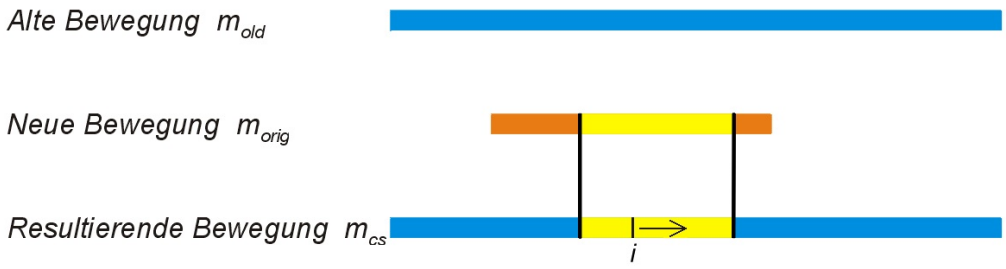


Abbildung 4.10: Resultierende Bewegung mit c_f, c_t != 0, s_f, s_t = 0

Dieser Fall unterscheidet sich nur gering von Fall 1. Der einzige Unterschied ist, dass i nun, da c_f und c_t angegeben wurden, nicht mehr ganz m_{cs} durchläuft, sondern nur noch den Bereich $c_f <= i <= c_t$. (4.4) gilt hier genauso.

Fall 3: nur Scale Die ersten beiden Fälle lassen sich relativ einfach handhaben. Sobald allerdings der Scale-Parameter dazukommt (Fälle 3 & 4), muss berücksichtigt werden, dass die skalierte Bewegung (m_{cs}) länger oder kürzer sein kann als die Originalbewegung (m_{orig}). Das bedeutet, wir müssen, bevor wir die alte (m_{old}) und die skalierte Bewegung (m_{cs}) zusammenkopieren, irgendwie aus m_{orig} m_{cs} ermitteln. Hierfür bietet es sich an, wieder eine Formel zu verwenden, die hier aber etwas komplizierter ist als die Formel oben. Wie sieht nun die Formel für diesen Fall aus? Schauen wir uns an, was wir wollen:

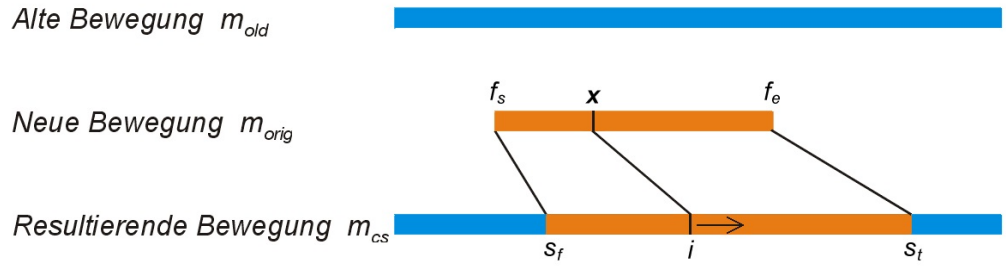


Abbildung 4.11: Resultierende Bewegung mit c_f, c_t = 0, s_f, s_t != 0

In Abbildung 4.11 fällt auf, dass wir den *Strahlensatz* verwenden können. Wir kennen f_s, f_e, s_f, s_t und i. Wir suchen x. Nach dem Strahlensatz ist nun

$$\frac{x - f_s}{f_e - f_s} = \frac{i - s_f}{s_t - s_f} \tag{4.5}$$

Aufgelöst nach x ergibt sich daraus

$$x = m_{orig}(i) = f_s + \frac{i - s_f}{s_t - s_f} \cdot (f_e - f_s) \tag{4.6}$$

Dies ist die Formel, die wir gesucht haben. Damit können wir nun für jedes Frame im skalierten Bereich von m_{cs} das entsprechende Frame in m_{orig} ermitteln. Genau so wird diese Formel auch im Code unseres Plugins verwendet.

Fall 4: Cut & Scale Der vierte Fall unterscheidet sich vom dritten dadurch, dass zusätzlich zu s_f und s_t auch noch Werte für c_f und c_t angegeben wurden. Ein Blick auf die neue Situation verrät uns aber schnell, wie (4.6) für diesen Fall anzupassen ist.

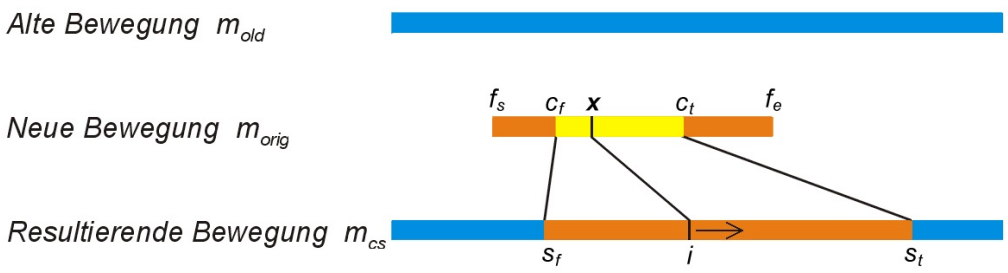


Abbildung 4.12: Resultierende Bewegung mit c_f, c_t != 0, s_f, s_t != 0

Da nun c_f und c_t angegeben sind, müssen wir nicht mehr die ganze neue Bewegung auf den durch s_f, s_t angegebenen Bereich skalieren, sondern nur noch das Stück zwischen c_f und c_t. Daraus folgt, dass wir in (4.6) einfach die Werte für f_s und f_e durch c_f und c_t ersetzen müssen.
Damit erhalten wir

$$\frac{x - c_f}{c_t - c_f} = \frac{i - s_f}{s_t - s_f} \tag{4.7}$$

Wiederum aufgelöst nach x ergibt sich

$$x = m_{orig}(i) = c_f + \frac{i - s_f}{s_t - s_f} \cdot (c_t - c_f) \tag{4.8}$$

4.2.2.5 Blend-Modul

Mit dem Blend-Modul ist es möglich, Bewegungen ineinander überzublenden. Dazu analysiert das Modul die alte und die neu zu importierende Bewegung und

stellt fest, in welchen Frames sich die beiden Bewegungen überschneiden. Denn an diesen Stellen muss geblendet werden. Es können allerdings höchstens zwei solcher Stellen vorhanden sein, nämlich am Anfang und am Ende der neuen Bewegung (Abb. 4.12 und Abb. 4.13). Es gibt einen Spezialfall, der im praktischen Einsatz häufig auftritt: Zwei Bewegungen, die hintereinander geschnitten und an der dabei entstehenden Grenze geblendet werden sollen (Abb. 4.14). Außerdem kann es vorkommen, dass sich keine Frames der alten und neuen Bewegung überschneiden (Abb. 4.15).

Abbildung 4.13: Blending am Anfang

Abbildung 4.14: Blending am Ende

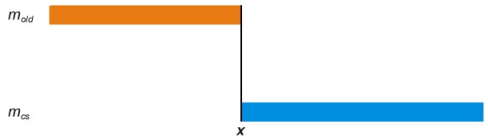


Abbildung 4.15: Blending in der Mitte

Abbildung 4.16: Blending in Lücke

Das Plugin bestimmt den Wert x in den obigen Abbildungen, also die Framenummer(n), in der/denen sich die beiden Bewegungen überlagern, indem es das erste und das letzte Frame der neuen Bewegung überprüft. Sei $i = f_{s,cs}$, also das erste Frame der neuen skalierten Bewegung. Dann wird geprüft, ob $m_{old}(i-1)$ vorhanden ist. Ist dies der Fall, müssen wir am Beginn von m_{cs} blenden (Fall 1, Abb. 4.12). i wird deshalb in einem Array zur späteren Weiterverwendung gespeichert. Sei nun $i = f_{e,cs}$, dann wird geprüft, ob $m_{old}(i+1)$ vorhanden ist. Ist dies der Fall, müssen wir am Ende der neuen Bewegung blenden (Fall 2, Abb. 4.13). Dieses i wird wiederum auf das Array gelegt.
Der dritte Blendfall, mit dem Blending in der Mitte, wird bei diesem Verfahren ebenfalls ermittelt.
Tritt der vierte Fall (keine gemeinsamen Frames) auf, prüft das Plugin, ob der Benutzer eine Blendtiefe b angegeben hat. Wenn nicht, bleibt während der Lücke (also in $\{f_{e,old}+1, f_{s,cs}-1\}$ die letzte Pose der alten Bewegung stehen. Wurde

eine Blendtiefe angegeben, überlässt das Plugin Maya während dieser Lücke die Kontrolle. Maya führt dann automatisch ein weiches Blending in diesem Bereich durch.
Der Anwender kann über das Interface eine Blendtiefe angeben (s. Abb. 4.5, Eingabefeld *blend depth*). Dieser Wert bestimmt, über wie viele Frames sich das Blending auf der rechten und linken Seite der soeben ermittelten Blendframes erstreckt, d.h. wenn i eines der ermittelten Frames ist, an denen sich die beiden Bewegungen überschneiden, und $b = 10$ die vom Anwender gewünschte Blendtiefe, dann verläuft das Blending im Intervall $\{i - b, i + b\}$.
Wurde dieser Bereich ermittelt, wird der Blendparameter u bestimmt als

$$u = \frac{1}{2 \cdot b + 2} \tag{4.9}$$

In dieser Formel werden im Nenner 2 addiert, da das Frame in der Mitte zwischen $i - b$ und $i + b$ dazukommt und dann wird nochmal 1 addiert, um den zur richtigen Weiterverarbeitung benötigten Wert für u zu erhalten.

Sind all diese Daten vorhanden, wird ein lineares Überblenden durchgeführt mit (Fall 1)

$$m_{blend}(i) = (1 - u) \cdot m_{old}(i) + u \cdot m_{cs}(i) \tag{4.10}$$

bzw. mit (Fall 2)

$$m_{blend}(i) = (1 - u) \cdot m_{cs}(i) + u \cdot m_{old}(i) \tag{4.11}$$

Die angegebene Blendtiefe b muss auch vom Cut/Scale-Modul beachtet werden. Deshalb durchläuft der Parameter i bei der Erstellung von m_{cs} nicht mehr nur den Bereich $\{f_{s,real}, f_{e,real}\}$, sondern jetzt $\{f_{s,real} - b, f_{e,real} + b\}$. Diese neuen Werte am Anfang und am Ende von m_{cs} müssen bei der Erstellung von m_{cs} aus m_{orig} (s. Abschnitt *Cut/Scale-Modul*) ebenfalls berücksichtigt werden. Dies ist aber kein großes Problem, da die dazu verwendeten Formeln (4.6) bzw. (4.8) auch mit dem um b erweiterten Bereich angewendet werden können. Abbildung 4.17 zeigt ein Beispiel:

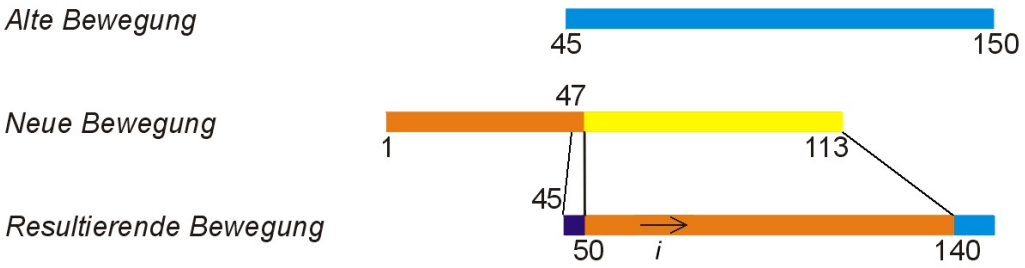


Abbildung 4.17: Konkretes Blend-Beispiel

Die alte Bewegung m_{old} läuft von Frame 45-150, die neue Bewegung m_{orig} von Frame 1-113. Von dieser Bewegung soll der Teil von $c_f = 50$ bis $c_t = 113$ ausgeschnitten und auf den Bereich von $s_f = 50$ bis $s_t = 140$ skaliert werden. Die Blendtiefe ist $b = 5$. Wie im vorherigen Abschnitt erklärt, durchläuft i m_{cs} nun von Frame 45 an, da 5 Frames für das Blending am Anfang dazukommen (in der Abbildung dunkelblau dargestellt). Durch Verwendung von (4.8) erhält man für alle Frames in diesem Bereich die dazugehörigen Frames von m_{orig}. In unserem Beispiel sind das die Frames 47-113. Diese werden, wie man sieht, auf den Bereich 45-140 aufskaliert. Das Blending verläuft immer auf beiden Seiten der Überschneidungsgrenze $g_1 = 50$, also im Intervall $\{g_1 - b, g_1 + b\}$ oder konkret von Frame 45-55.

An diesem Beispiel kann man sich sehr schön zwei Probleme veranschaulichen, die auftreten können und die vom Plugin automatisch gehandhabt werden. Zum einen könnte es passieren, dass m_{orig} nicht Frames von 45-150 enthält, sondern z.B. nur von Frame 48-150. Dann kann natürlich kein Blend von Frame 45-55 ausgeführt werden, sondern nur einer von Frame 48-55. Das zweite Problem ist, dass in diesem Beispiel ein weiterer Blend an der Grenze $g_2 = 140$ durgeführt werden müßte. Dies ist aber auch nur eingeschränkt möglich, da dieses Blending ja von $g_2 - b$ bis $g_2 + b$ verlaufen müßte, also von Frame 135-145. Das Frame 113 aus m_{orig} wird auf Frame 140 in m_{cs} aufskaliert. Allerdings gibt es keine Frames mehr in m_{orig}, um den Bereich von 141-145 zu füllen.

Das Plugin überprüft m_{old} und m_{orig} deshalb darauf, ob die beiden Bewegungen genügend Frames enthalten. Wenn nicht, führt es eine Analyse durch, um doch noch den Blend mit der maximal möglichen Anzahl an Frames auszuführen. Im

ersten Problemfall (m_{old} beginnt bei Frame 48) führt es einen Blend von Frame 48-55 durch. Im zweiten Problemfall einen von Frame 135-140. In solchen Fällen muss die Formel für die Berechnung des Blendparameters u (4.9) angepaßt werden. Die Anzahl der Frames auf jeder Seite werden getrennt ermittelt (b_{left} und b_{right}) und dann die jeweilige Anzahl in die Formel eingesetzt.
Für unsere beiden Problemfälle ergibt sich im ersten u zu

$$u = \frac{1}{b_{left} + b_{right} + 2} = \frac{1}{2 + 5 + 2} = \frac{1}{9} \tag{4.12}$$

im zweiten ist

$$u = \frac{1}{b_{left} + b_{right} + 2} = \frac{1}{5 + 0 + 2} = \frac{1}{7} \tag{4.13}$$

Durch eine kleine Anpassung von (4.8) kann das Plugin ganz schnell ermitteln, welches Frame m_{orig} enthalten muss, um den gewünschten Blend vollständig auszuführen. Im obigen Beispiel ist dies Frame 47. Die Formel, die dieses Frame am Anfang von m_{orig} ermittelt, lautet:

$$b_{startframe} = c_f + \frac{s_t - s_f + b}{s_t - s_f} \cdot (c_t - c_f) \tag{4.14}$$

Für das Ende von m_{orig} verwendet das Plugin folgende Formel:

$$b_{endframe} = c_t - \frac{s_t - s_f + b}{s_t - s_f} \cdot (c_t - c_f) \tag{4.15}$$

Stellt das Plugin fest, dass die so berechneten Frames nicht in den Dateien vorhanden sind, berechnet es die neue Blendtiefe am Anfang von m_{cs} per

$$b_{neu,start} = \frac{c_f - b_{filestartframe}}{c_f - b_{startframe}} \cdot b \tag{4.16}$$

und am Ende von m_{cs} per

$$b_{neu,end} = \frac{b_{fileendframe} - c_t}{b_{endframe} - c_t} \cdot b \tag{4.17}$$

$b_{filestartframe}$ und $b_{fileendframe}$ wurden mit der Funktion *frameExists_file()* ermittelt. Die folgende Abbildung veranschaulicht für den Fall am Anfang, was hier geschieht.

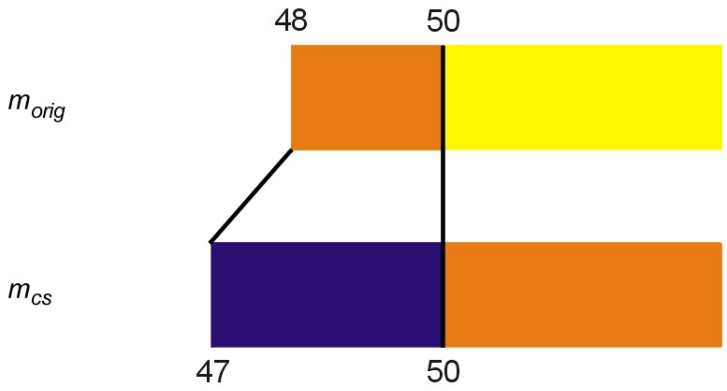


Abbildung 4.18: Rescaling von b

Um klarzumachen, was bezweckt wird, nehmen wir an, m_{orig} enthalte nicht die Frames 1-113 wie im Beispiel gerade eben, sondern nur die Frames 48-113. Alle anderen Werte bleiben gleich. Als erstes berechnen wir $b_{startframe}$ mit (4.14).

$$b_{startframe} = 113 + \frac{140 - 50 + 5}{140 - 50} \cdot (113 - 50) = 46.5 \tag{4.18}$$

Der Wert 46.5 wird gerundet auf 47. Dann ermittelt das Plugin per *frameExists_file()* den diesem Wert noch am nächsten liegenden, der in der aktuellen Datei (enthalten in m_{orig}) vorhanden ist. Bei uns ergibt das, wie gesagt, Frame 48 = $b_{filestartframe}$. Schließlich setzen wir alles in (4.16) ein und erhalten

$$b_{neu,start} = \frac{50 - 48}{50 - 47} \cdot 5 = 3.333... \tag{4.19}$$

wieder gerundet auf 3, als neue Blendtiefe. Der erste Faktor in (4.19) dient als Skalierungsfaktor für b. Mit diesem Ergebnis weiß das Plugin nun, dass der Parameter i zur Erzeugung von m_{cs} von Frame 47 an beginnen muss. Um Unklarheiten zu vermeiden, das Ergebnis von $b_{neu,start} = 3$ ist korrekt, obwohl m_{orig} ja nur 2 Frames vor Frame 50 enthält (48 und 49). Diese beiden werden aber aufskaliert, sodass eines dieser beiden Frames zweimal verwendet wird und somit stimmt es wieder genau. Das Plugin versucht also, wie man sieht, mit allen Mitteln noch den maximal möglichen Blend zu erzeugen.

4.2.2.6 Preserve-Modul

Das Preserve-Modul ermöglicht es, wenn im User Interface die Option *preserve old pos/rot* angewählt wurde, dass, wenn sich zwei Bewegungen überschneiden, an diesen Überschneidungsgrenzen kein Sprung zwischen der alten und der neuen Bewegung stattfindet, sondern, dass die neue Bewegung an der gleiche Stelle und mit der gleichen Rotation fortgesetzt wird, mit der die alte aufgehört hat.

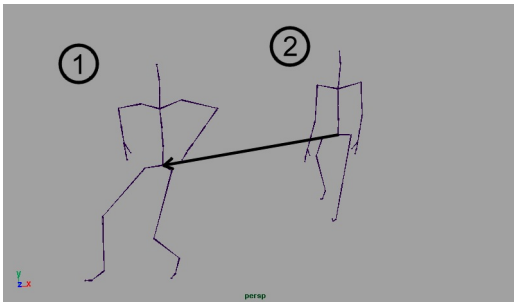


Abbildung 4.19: Szene ohne preserve

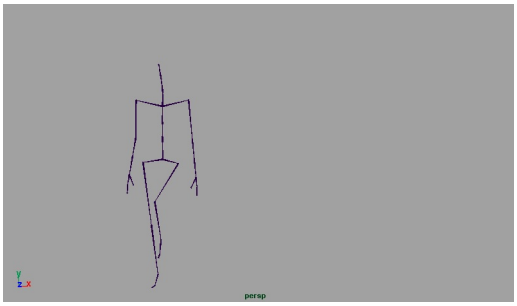


Abbildung 4.20: neue Szene mit preserve

Dieses Modul verwendet genau die gleichen Framegrenzen, die bereits im Blend-Modul ermittelt wurden. Um die gewünschte Funktionalität zu erhalten, gibt es zwei Dinge zu tun. Zum einen muss an den 0-2 maximal möglichen Framegrenzen, die auftreten können, die Translation und die Rotation der jeweils vorhergehenden Bewegung gespeichert werden. Der ganze Prozess muss allerdings nur für den Root-Knochen durchgeführt werden, da alle anderen Knochen diesem hierarchisch untergeordnet sind und somit automatisch mitgedreht und -verschoben werden. Angenommen, wir hätten eine Szene, in der es zwei Überschneidungsgrenzen g_1 und g_2 gibt. Dann müssen wir die Rotation und die Translation in den Frames $g_1 - 1$ und g_2 speichern, da diese jeweils die letzten Frames der vorherigen Bewegung darstellen. Zum anderen muss bei der Erstellung der kombinierten Bewegung beachtet werden, dass jedesmal, wenn das Frame mit der Nummer i, das gerade erzeugt wird, größer ist als eine der ermittelten Grenzen ($i > g_1 - 1$ oder $i > g_2$), die zuvor gespeicherte Rotation und Translation an dieser Grenze zu allen nachfolgenden Frame addiert wird. Bei der ersten Grenze werden die Werte von $g_1 - 1$ gespeichert, da die Framegrenzen als erstes und letztes Frame der *aktuellen* Bewegung definiert sind und wir deshalb das Frame,

das eins weiter vorne liegt, speichern müssen. Es gilt also für die Translation T

$$T_{i,neu} = T_i + \sum_{j=1}^{n} T_{g_j} \tag{4.20}$$

für $i > g_j$ und n=Anzahl der Grenzen ($0 <= n <= 2$). Das Summenzeichen wird verwendet, weil für alle vorkommenden Framegrenzen alle die dort gespeicherten Werte zur aktuellen Translation addiert werden müssen, um das gewünschte Ergebnis zu erhalten.

4.2.2.7 Interpolate-Modul

Das Interpolate-Modul ist ein Post-Processing-Schritt, der automatisch auf alle neu erstellten Bewegungskurven angewendet wird, wenn

$$s_t - s_f > f_{end,real} - f_{start,real} \tag{4.21}$$

erfüllt ist, d.h. wenn die skalierte Bewegung länger ist als die Originalbewegung und sie somit gleiche Frames mehrfach enthält. Abbildung 4.21 zeigt eine solche Animationskurve ohne Interpolation. Abbildung 4.22 zeigt die gleiche Kurve mit Interpolation.

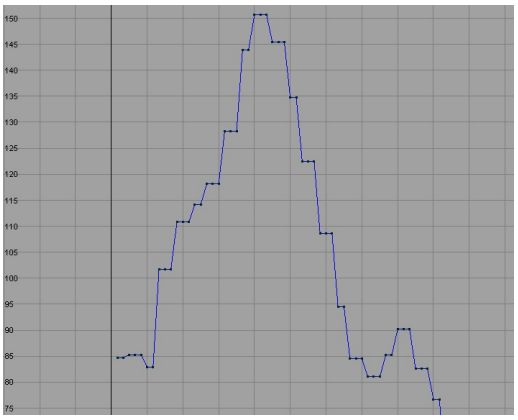


Abbildung 4.21: Kurve ohne Interpolation

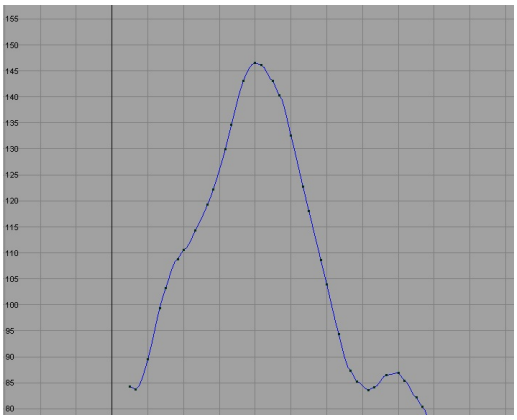


Abbildung 4.22: Kurve mit Interpolation

Ohne Interpolation macht sich die Skalierung in der Animation selbst bemerkbar durch ein deutlich sichtbares Ruckeln. Verwendet man dagegen die interpolierte

Version, sind die Bewegungen wieder absolut weich.
Der Interpolationsalgorithmus ermittelt für jede Kurve die Intervalle, wo mehrere gleiche Werte hintereinander auftreten. Er bestimmt die Länge l_{gleich} dieser Stücke und addiert zu dieser Länge 2 dazu, da die Interpolation zwischen den l_{gleich} gleichen und den beiden angrenzenden Werten stattfindet.

$$l_{neu} = l_{gleich} + 2 \tag{4.22}$$

Diese l_{neu} Werte werden auf ein Array gelegt. Zuerst wird die Steigung m zwischen den beiden Grenzwerten berechnet.

$$m = m_{cs}(l_{neu}) - m_{cs}(1) \tag{4.23}$$

Außerdem benötigen wir einen Faktor u ($0 <= u <= 1$). Dieser wird berechnet als

$$u = c \cdot \frac{1}{l_{neu} - 1} \qquad \text{mit } 0 <= c <= l_{neu} - 1 \tag{4.24}$$

Dann wird die Kurve linear interpoliert, indem für jeden der gleichen Werte ein interpolierter Wert i berechnet wird per

$$i(c) = m_{cs}(1) + u \cdot m \tag{4.25}$$

Die Abbildungen 4.23 und 4.24 zeigen diesen Prozess im Detail.

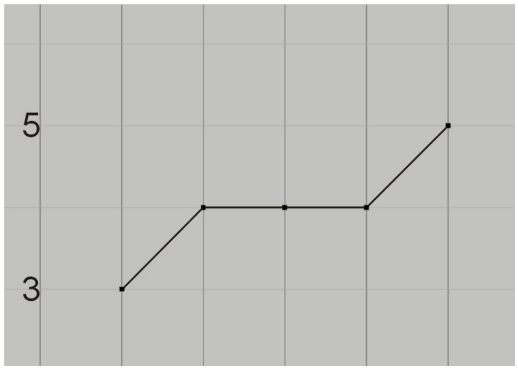


Abbildung 4.23: Kurve ohne Interpolation (Detail)

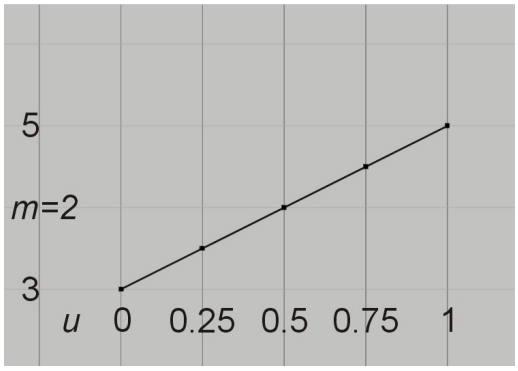


Abbildung 4.24: Kurve mit Interpolation (Detail)

4.3 Bewegungs-Filter

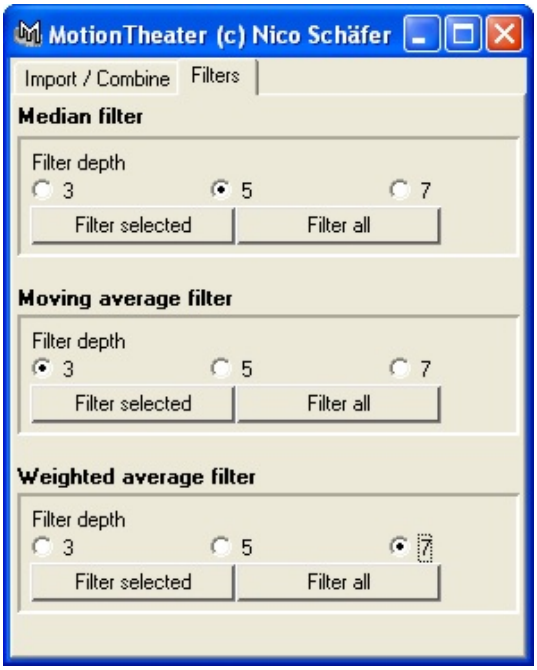


Abbildung 4.25: MotionTheater Filter-Interface

MotionTheater stellt dem Anwender verschiedene digitale Filter zur Glättung der Bewegungskurven zur Verfügung. Diese können dazu benutzt werden, um Rauschen (hochfrequente Störungen im Bewegungssignal), die durch Messungenauigkeiten entstehen, zu entfernen und ebenfalls um das gesamte Signal bei Bedarf zu glätten. Motion Capture-Signale haben eine große Ähnlichkeit mit Audiosignalen und können wie diese behandelt werden. Viele in der Audiotechnik verwendeten Filter können auch auf Bewegungssignale angewandt werden. Eine Arbeit, die dies sehr schön zeigt, ist [5].
MotionTheater bietet drei verschiedene Arten von Filtern an: einen *Median-Filter*, einen *Mittelwert-Filter* und einen *gewichteten Mittelwert-Filter*. In Abbildung 4.25 sieht man das Interface für diese Filter. Für jeden Filter läßt sich eine Filtertiefe von 3, 5, oder 7 einstellen. Die Filtertiefe bestimmt, wie viele Werte aus der Bewegungskurve berücksichtigt werden, um den aktuell zu filternden Wert zu erhalten. Genau wie beim *MotionCombiner* ist es hier möglich, die Filter entweder auf einzelne Knochen anzuwenden, die man zuvor im Maya *Hypergraph* ausgewählt hat

(Button *Filter selected*), oder auf das gesamte Skelett (Button *Filter all*). Abbildung 4.26 und 4.27 zeigen eine Bewegungskurve in Maya in der ungefilterten (oben) und in der gefilterten Version (unten).

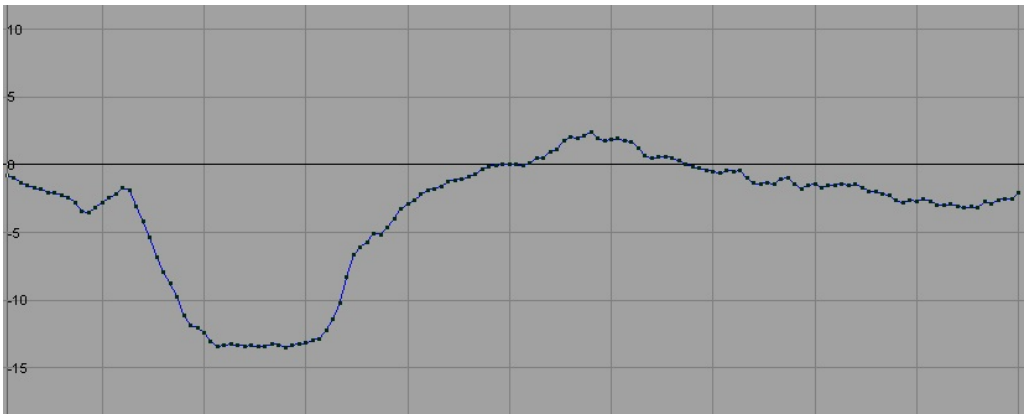


Abbildung 4.26: Ungefilterte Bewegungskurve

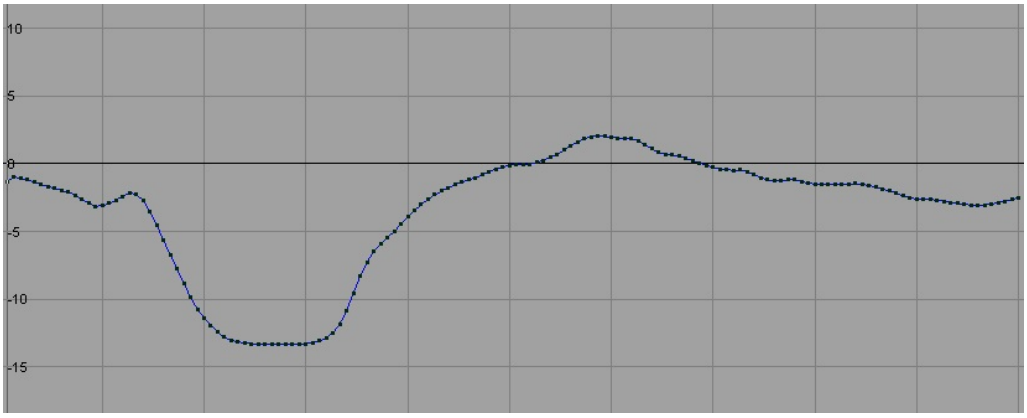


Abbildung 4.27: Gefilterte Bewegungskurve

Ein digitaler Filter bearbeitet ein Signal, indem er den aktuellen Wert, der gefiltert werden soll, und eine in der Größe festlegbare Umgebung um diesen Wert herum untersucht. Aus diesen Werten berechnet er den neuen gefilterten Wert. Wie das jeweils genau funktioniert, wird detailliert in den Abschnitten 4.3.1, 4.3.2 und 4.3.3 erklärt.

Die Filter als Maya-Plugin in C++ zu implementieren, bietet den Vorteil, dass die Ausführungsgeschwindigkeit wesentlich höher ist als bei einer Implementierung in der Maya-eigenen Skriptsprache MEL. Zum Vergleich werden hier die Werte aus [3] verwendet, deren Arbeit eine MEL-Implementierung dieser Filter enthält.

Um die Ausführungsgeschwindigkeit vergleichen zu können, wurde genau wie in [3] eine Szene mit 24.000 Werten benutzt. Hiller/Bomm verwenden eine Szene mit 40 Markern und 200 Frames. In deren Arbeit wurden die Positionen der Marker im Raum in Maya importiert und gefiltert. *MotionTheater* arbeitet nicht mit Markern, sondern mit Gelenkrotationen. Sowohl die Positionen der Marker im Raum, als auch die Gelenkrotationen bestehen jedoch aus drei float-Werten. Unsere Testszene besteht aus 30 Gelenken und dafür 267 Frames. Somit werden hier ebenfalls ca. 24.000 Werte gefiltert. Die Filtertiefe ist $b = 3$ wie beim Test der MEL-Implementierung in [3].

Filtertyp	*Hiller/Bomm*	*MotionTheater*
Mittelwert	1,99 s	0,66 s
gewichteter Mittelwert	2,05 s	0,64 s
Median	2,08 s	0,59 s

Die C++-Implementierung des Plugins braucht im Durchschnitt etwas weniger als ein Drittel der Zeit, die das MEL-Skript benötigt. Besonders bei langen Motion Capture-Szenen mit vielen Frames, wie sie im Produktionsalltag häufig vorkommen, unter anderem deshalb, weil oft mit 120 Hz (120 Bilder/sek.) gecaptured wird, macht sich dieser Geschwindigkeitsvorteil schnell bezahlt.

4.3.1 Median-Filter

Der Median-Filter ist sehr gut geeignet, um sogenannte *Peaks* (durch Messungenauigkeiten entstandene Spitzen im Bewegungssignal) und durch ausgefallene Marker entstandene Fehler zu beseitigen. Der Median erhält, im Vergleich zu den Mittelwertfiltern, die Originalform des Signals am besten. Außerdem erzeugt er keine neuen Werte, sondern verwendet nur die Werte, die schon im Signal vorhanden sind. Der Nachteil des Median ist die Sortierung der Werte, die er verwendet (siehe nächster Abschnitt) und die für jeden zu filternden Wert durchgeführt werden muss. Dadurch wird der Median bei großen Filtertiefen sehr langsam. In den Abbildungen 4.28-4.31 sieht man schön, wie ein Peak aus dem Signal

entfernt wird (oben). Flanken, die Teil des Originalsignals sind, bleiben erhalten (unten).

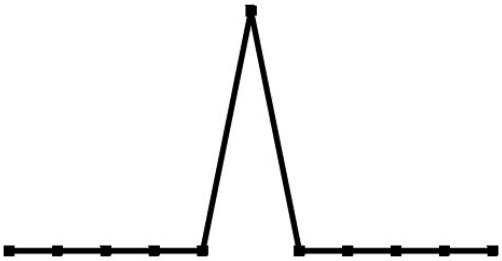


Abbildung 4.28: Bewegung 1 ungefiltert

Abbildung 4.29: Bewegung 1 Median-gefiltert

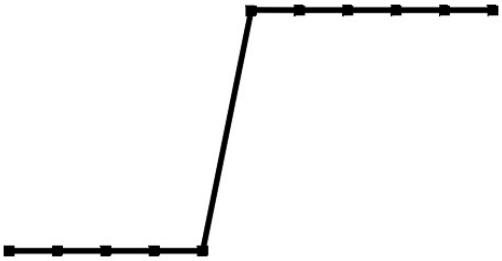


Abbildung 4.30: Bewegung 2 ungefiltert

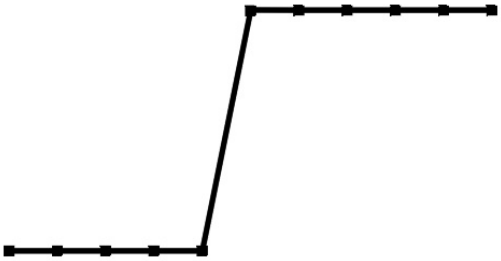


Abbildung 4.31: Bewegung 2 Median-gefiltert

Der Median-Filter erreicht dies, indem er alle Werte frameweise bearbeitet und den aktuell zu untersuchenden Wert plus die der Filtertiefe entsprechende Anzahl Frames links und rechts davon in ein Array lädt. Dieses Array wird dann nach aufsteigenden Werten sortiert und der mittlere dieser Werte wird als Wert für den aktuellen Abtastpunkt genommen.

4.3.2 Mittelwert-Filter

Der Mittelwert-Filter ist am besten dafür geeignet, *Rauschen* aus einem Bewegungssignal zu entfernen. Zusätzlich erhält man durch ihn weichere Animationen, da er das Bewegungssignal glättet. Der Nachteil des Mittelwert-Filters ist jedoch, dass er neue Werte erzeugt, die ursprünglich nicht im Signal enthalten waren. Ist die Filtertiefe zu groß oder wird der Filter zu oft angewandt, kann es deshalb schnell zu unterwünschten Effekten kommen. In den Abbildungen 4.32-4.35 sehen wir links die gleichen Ausgangssignale wie gerade eben und rechts daneben

die mit dem Mittelwert-Filter geglätteten Signale. Peaks entfernt der Filter nicht ganz so zuverlässig wie der Median, dafür erzeugt er bei Signal 2 einen weicheren Übergang als der Median, der bei diesem Filter allerdings linear ist (Gerade).

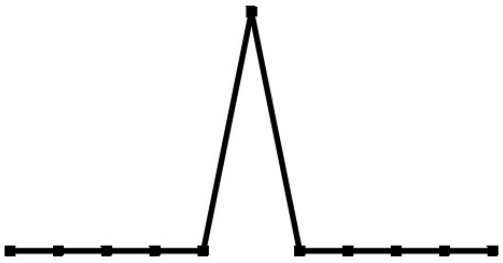


Abbildung 4.32: Bewegung 1 ungefiltert

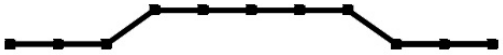


Abbildung 4.33: Bewegung 1 Mittelwert-gefiltert

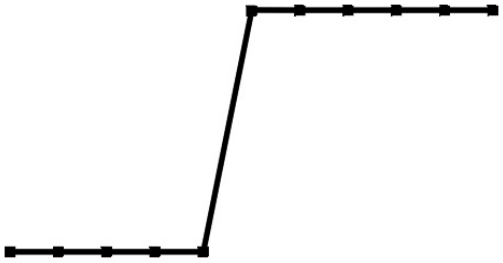


Abbildung 4.34: Bewegung 2 ungefiltert

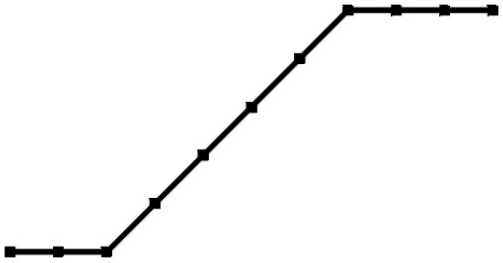


Abbildung 4.35: Bewegung 2 Mittelwert-gefiltert

Der Mittelwertfilter berechnet den gefilterten Wert, indem er Werte aus der Umgebung des zu filternden Wertes addiert und durch deren Anzahl (entspricht der Filtertiefe) dividiert. Der gefilterte Wert y ergibt sich zu

$$y(i) = \frac{1}{n} \sum_{j=-\frac{n}{2}+\frac{1}{2}}^{\frac{n}{2}-\frac{1}{2}} x(i+j) \tag{4.26}$$

n ist hierbei die Filtertiefe, was der Anzahl der zu untersuchenden Werte entspricht. n kann nur ungerade positive Werte annehmen. Deshalb bietet das Plugin nur Filtertiefen von 3, 5 und 7 an. i läuft über alle zu filternden Frames.

4.3.3 Gewichteter Mittelwert-Filter

Der gewichtete Mittelwertfilter ist eine verbesserte Version des eben vorgestellten normalen Mittelwertfilters. Der gewöhnliche Mittelwertfilter behandelt alle Wer-

te, die er untersucht, gleich. Der gewichtete Mittelwertfilter gewichtet den aktuellen mittleren Wert am stärksten und die links und rechts davon liegenden Werte schwächer, je weiter sie vom Zentrum entfernt liegen. Der Vorteil dieser Methode ist, dass man noch weichere Bewegungen als mit dem gewöhnlichen Mittelwertfilter erhält (da Kurve anstatt Gerade erzeugt wird) und das Originalsignal wegen der Gewichtung besser erhalten bleibt. Dies sieht man schön auf den Abbildungen 4.36-4.39.

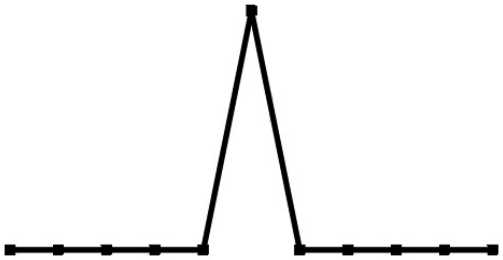


Abbildung 4.36: Bewegung 1 ungefiltert

Abbildung 4.37: Bewegung 1 gew. Mittelwert-gefiltert

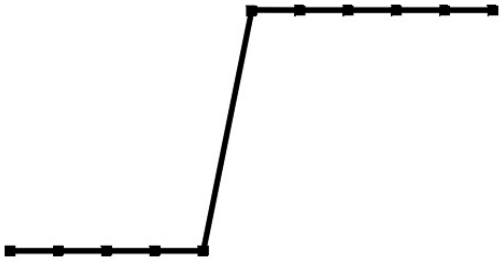


Abbildung 4.38: Bewegung 2 ungefiltert

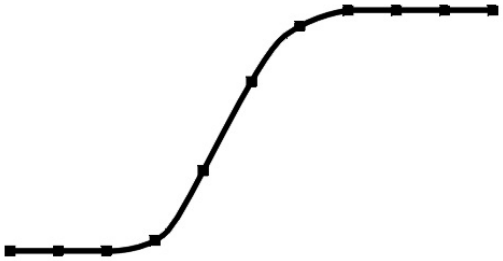


Abbildung 4.39: Bewegung 2 gew. Mittelwert-gefiltert

Es gibt verschiedene Möglichkeiten, die Gewichte für die einzelnen Werte zu finden. Unser Plugin verwendet dazu das *Pascal'sche Dreieck*. Die Gewichte seien hier mit b bezeichnet. Das Plugin erzeugt diese als Array. Wir schauen uns die Berechnung der Gewichte an einem konkreten Beispiel an. Angenommen, die vom Benutzer spezifizierte Filtertiefe sei $t = 5$. Dann erzeugt das Programm die fünfte Stufe des Pascal'schen Dreiecks. Diese sieht bekanntlich so aus: $1 \quad 4 \quad 6 \quad 4 \quad 1$. Diese fünf Werte werden addiert ($= 16$) und die Gewichte erstellt, indem jeder Wert im Pascal'schen Dreieck durch diese Summe geteilt wird. Wir erhalten also

$b = \{\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}\}$. Zur Überprüfung muss dabei

$$\sum_{k=1}^{5} b_k = 1 \tag{4.27}$$

gelten, ansonsten wird das Signal verfälscht.
Mit diesen Werten ermittelt das Programm den endgültigen Wert y als

$$y(i) = \sum_{j=-\frac{n}{2}+\frac{1}{2}}^{\frac{n}{2}-\frac{1}{2}} b_k \cdot x(i+j) \tag{4.28}$$

Kapitel 5

Tutorial

5.1 Installation des Plugins

1. Kopieren Sie das Plugin von der CD, die dieser Diplomarbeit beiliegt, in folgendes Verzeichnis: *C:\Programme\Maya5.0\bin\plug-ins*\. Wenn sich Ihre Maya-Installation an einem anderen Ort befindet, kopieren Sie es bitte in das entsprechende Verzeichnis.

2. Um das Plugin zu laden, wählen Sie in Maya unter *Window > Settings/Preferences > Plug-in Manager...*

3. Im sich öffnenden Fenster wird nun das Plugin aufgelistet, wenn es zuvor in das obige Verzeichnis kopiert wurde. Aktivieren Sie das Kästchen *loaded* neben MotionTheater.mll (Abbildung 5.1) bzw. *auto load*, wenn Sie möchten, dass *MotionTheater* automatisch geladen wird. Sie müssen dann die Schritte 2 & 3 nicht jedesmal, wenn Sie Maya starten, von Hand ausführen.

4. Klicken Sie auf *Close*. Das Plugin ist nun geladen und einsatzbereit.

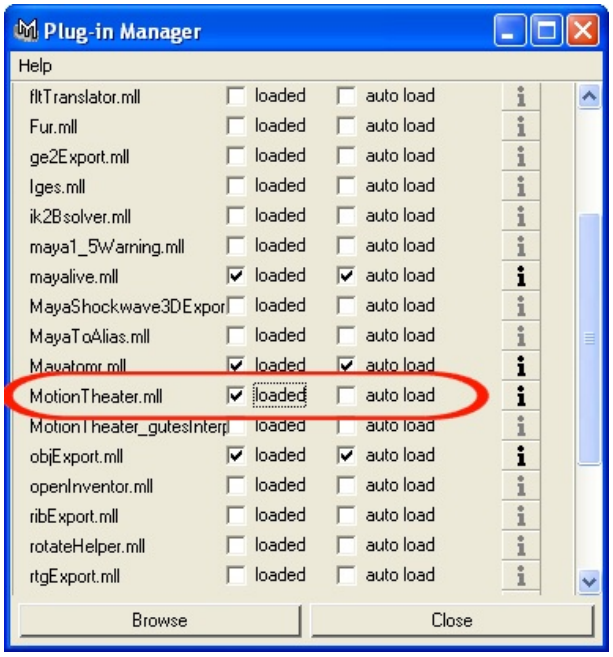


Abbildung 5.1: Maya Plug-in Manager

5.2 Import der Bein-Bewegung

In diesem Tutorial erstellen wir eine kombinierte Bewegung, die aus Daten von zwei verschiedenen Bewegungsaufnahmen besteht, um die Funktionalität des *MotionCombiners* aufzuzeigen.

1. Rufen Sie das User Interface für *MotionTheater* auf, indem Sie in die Maya Command Line *source MotionTheater;* eingeben. Nach einem Druck auf <Return> erscheint das *MotionTheater*-UI (Abbildung 5.2).

2. Das UI besteht aus zwei Reitern: der erste enthält das Interface für den *MotionCombiner*, der zweite beinhaltet die Filter.

3. Als nächstes importieren wir über den Importer des Plugins eine ASF-Datei. Diese Datei enthält die erste Pose (meist eine T-Pose) des Skeletts. Klicken Sie auf den Button *Import .asf file (setup pose)*. Es erscheint ein Dateiauswahlfenster. Wechseln Sie darin auf die *MotionTheater*-CD und doppelkli-

cken Sie auf `oxford.asf`. Das Skelett wird geladen und erscheint in der Szene.

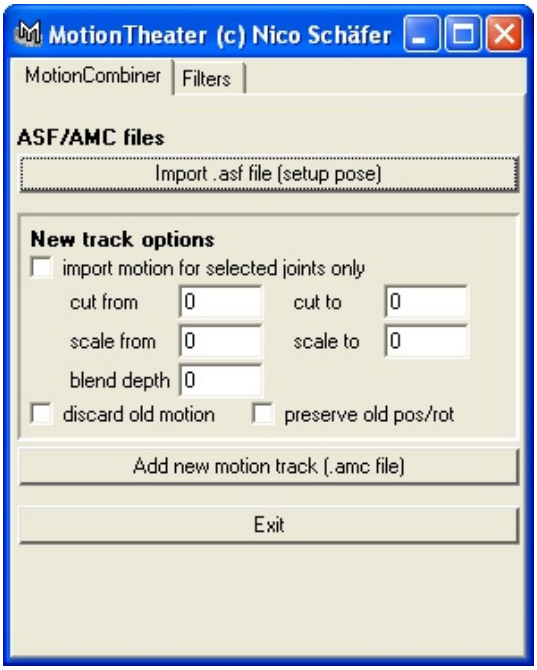


Abbildung 5.2: *MotionTheater* User Interface

4. Um die gesamte Szene betrachten zu können, klicken Sie in das 3D-Fenster von Maya und drücken die Taste 'A'.

5. Wir wollen eine Bewegung für die Beine importieren und zwei andere Bewegungen für den Oberkörper und die Arme. Um alle Beinknochen auszuwählen, öffnen Sie den Maya *Hypergraph* über *Window* > *Hypergraph...*. Im Hypergraph sind alle Knochen, aus denen das Skelett besteht, aufgelistet. Maximieren Sie ihn am besten, um genau erkennen zu können, wie die Knochen heißen. Drücken Sie dazu wiederum auf 'A', um alle Knochen im Hypergraph sehen zu können.

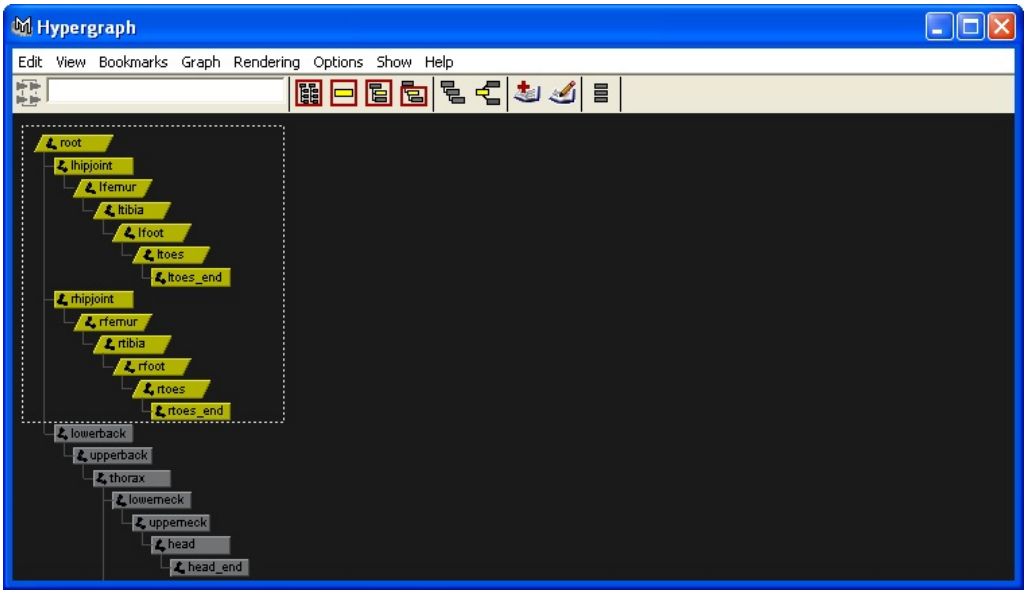


Abbildung 5.3: Knochenauswahl im Hypergraph (Beine)

6. Die Beine sind in der Hierarchie oben aufgelistet. Der oberste Knochen ist die *root*. Daran schließen sich zwei gleichlange Stränge an, die jeweils mit *lhipjoint* bzw. *rhipjoint* beginnen. Dies sind das linke und das rechte Hüftgelenk, worauf der Rest der Beine folgt. Wählen Sie alle Knochen von *root* bis einschließlich *rtoes_end* aus, indem Sie bei gedrückter linker Maustaste einen Rahmen darum aufziehen. Die ausgewählten Knochen erscheinen in gelb (Abbildung 5.3).

7. Da wir die erste Bewegung nur für die Beine importieren wollen, aktivieren Sie bitte im Feld *New track options* das Kontrollkästchen *import motion for selected joints only*.

8. Klicken Sie auf den Button *Add new motion track (.amc file)* im *MotionTheater*-Interface und wählen Sie im sich öffnenden Fenster von der CD die Datei `oxford.stepjump.amc` aus. Durch Doppelklick oder per Klick auf *Öffnen* tritt der AMC-Importer in Aktion und lädt die gesamte Bewegung, die in diesem File gespeichert ist.

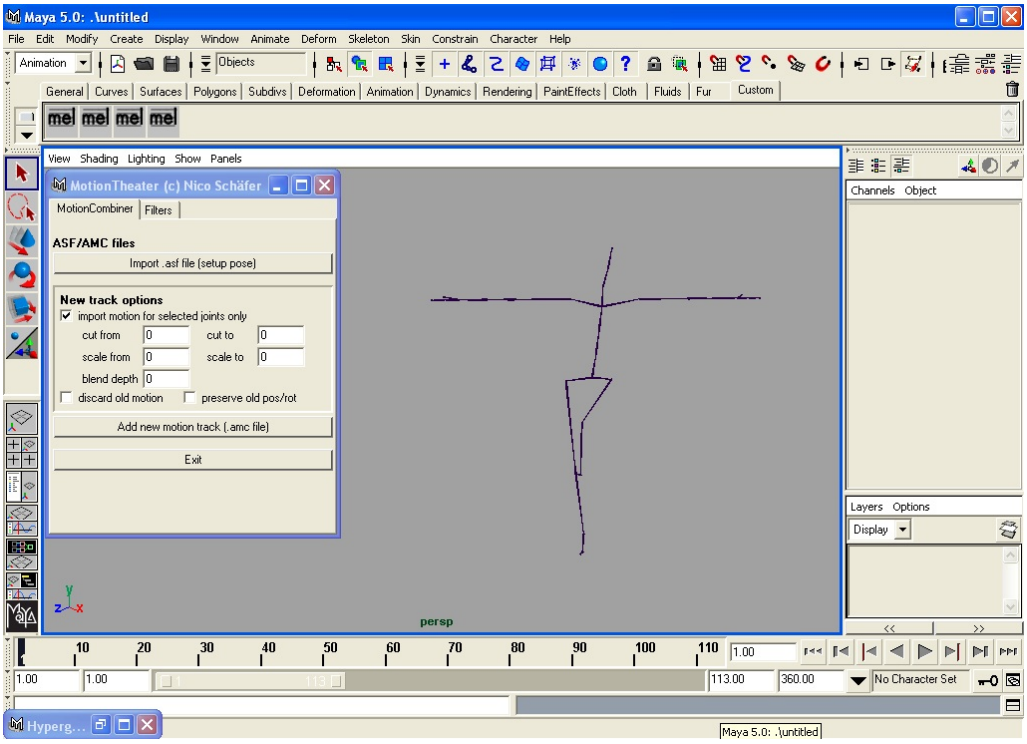


Abbildung 5.4: Importiertes AMC-File

9. Klicken Sie in Maya auf den *Play*-Button unten rechts und Sie sehen die Bewegung. Die Beine bewegen sich, Arme und Oberkörper stehen noch still, da wir die erste Bewegung ja nur für die Beine importiert haben.

5.3 Import der 1. Arm-Bewegung

Jetzt importieren wir für den Oberkörper und die Arme eine andere Bewegung.

1. Wählen Sie im Hypergraph alle Knochen aus, die bisher nicht ausgewählt waren. Das sind die Knochen für Oberkörper und Arme (unterer Teil der Hierarchie, Abbildung 5.5).

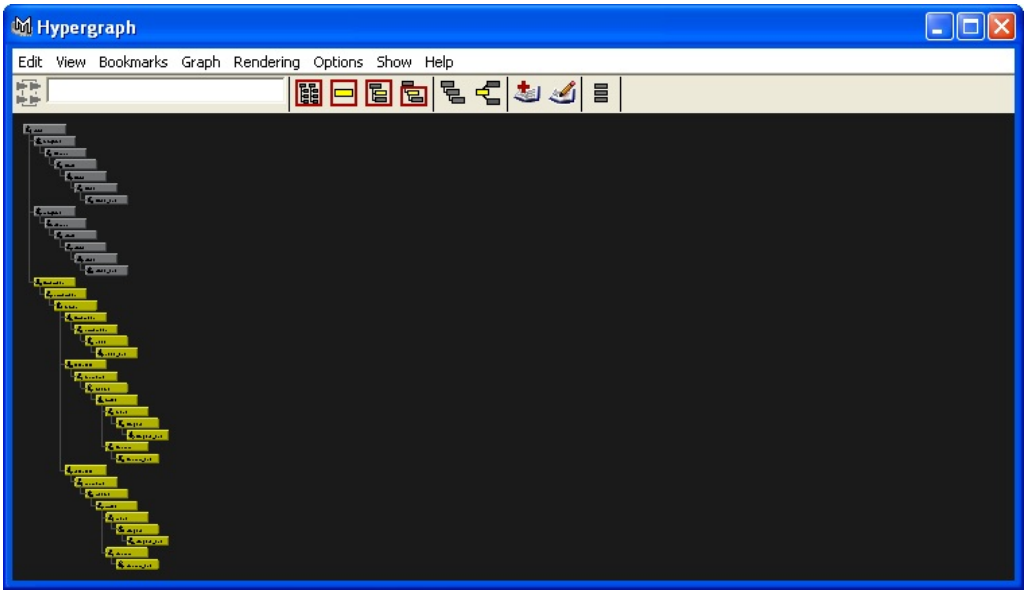


Abbildung 5.5: Knochenauswahl im Hypergraph (Oberkörper)

2. Geben Sie folgende Werte in die Eingabefelder des User Interfaces ein: *cut from*: 1, *cut to*: 55, *scale from*: 1, *scale to*: 66. Wir schneiden also aus der neu zu importierenden Bewegung die Files 1-55 aus und skalieren sie auf den Bereich Frame 1-66.

3. Klicken Sie wieder auf den Button *Add new motion track (.amc file)* und öffnen Sie im Dateiauswahlfenster die Datei `oxford.aljolson.amc`.

4. Schauen Sie sich die neu erstellte Bewegung an, indem Sie wieder den Play-Button aktivieren. Oberkörper und Arme bewegen sich nun in den Frames 1 bis 66. Ab Frame 67 stehen sie wieder still, da wir dafür noch keine Bewegung importiert haben.

5.4 Import der 2. Arm-Bewegung mit Blending

1. Als nächstes wollen wir für den restlichen Bereich (Frames 67-113) wieder die Originalarmbewegung importieren, damit diese während des Sprungs zur übrigen Bewegung paßt und natürlich wirkt. Geben Sie folgende Werte

in die Eingabefelder ein: *cut from*: 67, *cut to*: 113, *scale from*: 0, *scale to*: 0, *blend depth*: 10.

2. Klicken Sie auf *Add new motion track (.amc file)* und laden Sie dieses Mal wieder die Datei `oxford.stepjump.amc`.

3. Schauen Sie sich die neue Bewegung an und wie sich jetzt die Bewegung der Arme um Frame 66 herum durch das Blending zwischen der alten und der neuen Bewegung verändert hat. Das Blending wurde durch die Eingabe der Blendtiefe von 10 im UI vom Plugin automatisch berechnet.

4. Wenn Sie möchten, können Sie auch einmal in einer neuen Szene jeweils getrennt die Bewegungen `oxford.aljolson.amc` und `oxford.stepjump.amc` laden und sich besonders die Bewegungen der Arme in beiden Files im Bereich um Frame 66 herum anschauen, um zu sehen, wie diese ohne Blending aussehen.

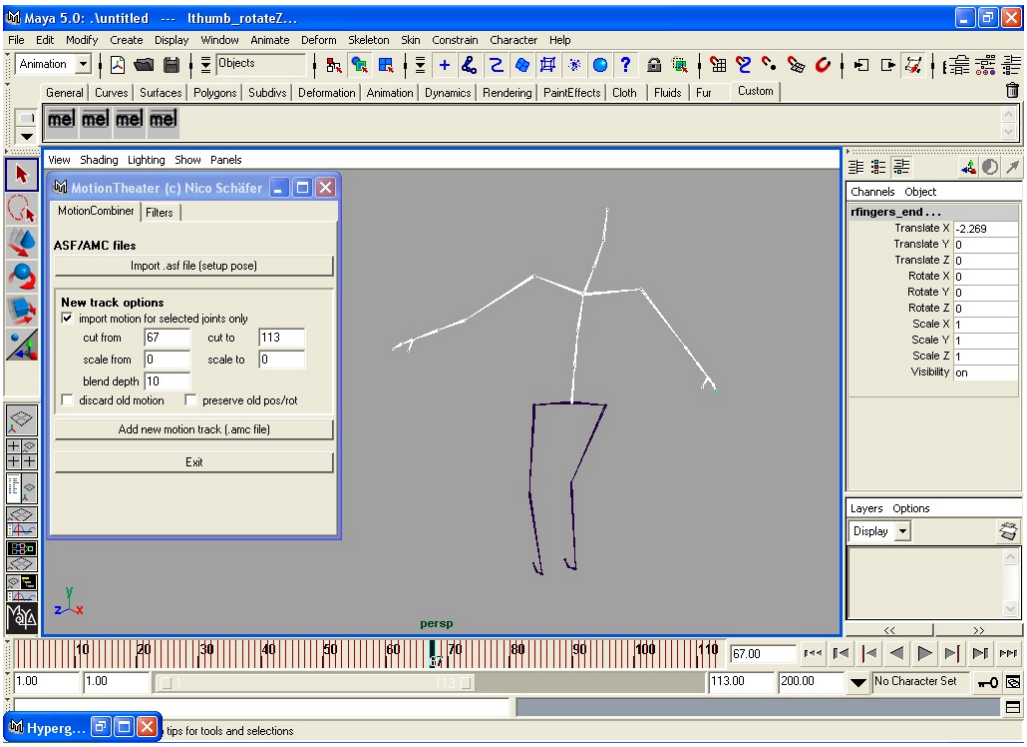


Abbildung 5.6: Armbewegung im Blending-Bereich

5. Speichern Sie die Szene ab.

5.5 Filterung der Armbewegung

Um die Funktionsweise der Filter zu demonstrieren, verwenden wir die soeben erstellte Szene und filtern die Armbewegung (das Winken).

1. Schauen Sie sich zuerst die Armbewegung an, wie sie momentan aussieht (*Play*-Button).

2. Wählen Sie im Hypergraph die Arme des Skelettes aus. Dies sind die Stränge, die mit `lclavicle` und `rclavicle` beginnen.

3. Wechseln Sie im *MotionTheater*-UI von der *MotionCombiner*-Karte zur *Filters*-Karte.

4. Stellen Sie im Feld des *Weighted average filter* (gewichteter Mittelwert) als Filtertiefe 7 ein.

5. Klicken Sie im selben Feld auf *Filter selected*, um wieder nur die ausgewählten Knochen (Arme) zu filtern. Sie sehen eine kleine Veränderung der Armhaltung. Dies zeigt an, dass der Filter vorgenommen hat. Zusätzlich wird es durch die Meldung *AnimCurves of selected joints filtered successfully* in der Command Feedback line bestätigt.

6. Schauen Sie sich die gefilterte Bewegung der Arme an und Sie werden bemerken, dass das Winken jetzt nicht mehr so stark ausgeprägt ist wie vor der Filterung. Der Filter glättet die Animationskurven und entfernt *Peaks*. Diese zeigen sich in der Bewegung als hohe Frequenzen = schnelle Bewegungen. Da diese durch den Filter reduziert wurden, fällt auch das Winken nicht mehr so stark aus.

7. Wenn Sie mit dem Filterergebnis nicht zufrieden sind oder einen anderen Filtertyp oder eine andere Filtertiefe ausprobieren möchten, drücken

Sie <Strg>+Z, um die Filterung rückgängig zu machen. Mit <Shift>+Z können Sie sie wiederherstellen. Sie können natürlich auch beliebig viele Filter nacheinander auf die Bewegung anwenden.

8. Damit sind wir am Ende des Tutorials angekommen. Speichern Sie die Szene noch einmal. Abbildung 5.7 zeigt einen Screenshot von der komplett bearbeiteten Bewegung. Wenn Sie wollen, können Sie nun mit dem Plugin experimentieren und andere Bewegungen beliebig kombinieren. Dazu finden Sie auf der CD noch zwei weitere Bewegungsfiles: `oxford.dancepirohette.amc` und `oxford.satnightfever.amc`. Viel Spaß!

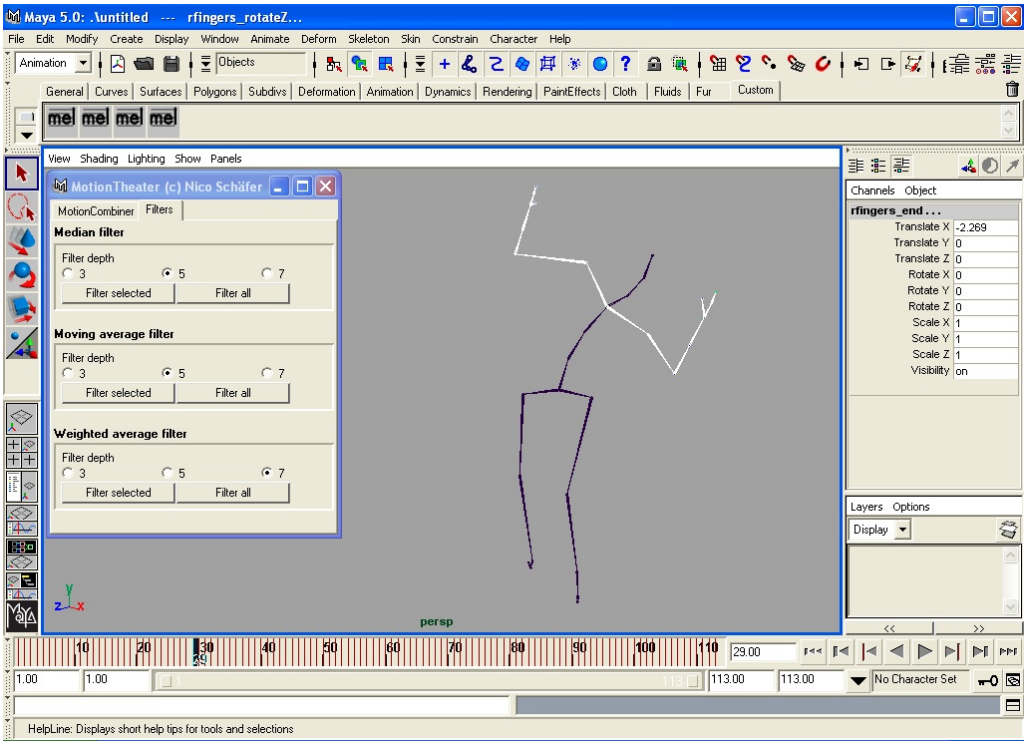


Abbildung 5.7: Die fertige Szene

Kapitel 6

Zusammenfassung und Ausblick

Diese Arbeit stellt die Vorarbeit für das an der HdM geplante Motion Capture-Datenbank-Projekt dar, das ab Obktober 2004 in Zusammenarbeit mit der Universität Bonn durchgeführt wird. Dafür wurde in dieser Diplomarbeit ein Plugin für die 3D-Software Maya erstellt, mit dem es möglich ist, Motion Capture-Daten im ASF/AMC-Format in Maya zu importieren und in beliebiger Weise zu kombinieren. Es ist möglich, die Bewegungen zu schneiden, zu skalieren, Bewegungen ineinander überzublenden und sie zu filtern.

Zusätzlich zu der hier implementierten Funktionalität gäbe es noch einige Bereiche, um die man das Plugin erweitern könnte. Das Wichtigste ist, Retargeting und Constraints einzubauen. *Retargeting* ist der Vorgang, die Bewegung, die von einem menschlichen Schauspieler aufgenommen wurde, auf einen im Computer modelierten Character zu übertragen, da dies sehr häufig gebraucht wird. Das Retargeting erfordert auch unerlässlicherweise *Constraints*, also Zwangsbedingungen, die man der Bewegung zuweisen kann. Dies ist nötig, da Original- und Zielskelett oft nicht die gleichen Proportionen haben. Deshalb muss man bei der Zielbewegung Constraints setzen können, um z.B. weiterhin sicherzustellen, dass der Character nicht mit seinen Beinen den Boden durchdringt / darüber schwebt oder bei einem Objekt, das er eigentlich ergreifen soll, ins Leere oder in das Objekt greift, da seine Arme kürzer bzw. länger sind als die des Schauspieler, der beim Capturing agierte.

Im Bereich der Filter wäre ein *Frequenzfilter* wünschenswert, mit dem man die

Bewegung in mehrere Frequenzbänder aufspalten kann. Man hat dann Bewegungskurven für hohe, mittlere und tiefe Frequenzen und kann diese einzeln per Schieberegler verstärken oder abschwächen. Verstärkt man z.B. die tiefen Frequenzen, verstärkt sich die gesamte Bewegung - der Character bewegt sich betonter. Schwächt man hohe Frequenzen ab, verschwindet eventuell vorhandenes Rauschen usw.
Für diese Arbeit war noch ein Tool angedacht, das aus Zeitgründen nicht mehr implementiert werden konnte - der *SkeletonAnalyzer*. Dieses Tool sollte dafür sorgen, dass jede beliebige Anzahl von Markern beim Motion Capturing mit jeder beliebigen Anzahl von Knochen eines Skelettes verwendet werden kann. Der *MotionAnalyzer* muss das gecapturete File und das gegebene Skelett darauf überprüfen, wie viele Marker/Knochen beide enthalten und dann Korrespondenzen zwischen diesen herstellen. Auch wenn beim Capturing sehr wenige Marker verwendet wurden, soll der *SkeletonAnalyzer* für das Zielskelett immer noch die maximal mögliche Anzahl an Daten aus den Markerdaten ermitteln, um die Bewegung so realistisch wie möglich zu machen.

Literaturverzeichnis

[1] Alias|Wavefront: *Maya 4 Online Library - Developer Guide*.

[2] Alias|Wavefront: *Maya 4 Online Library - MEL Command Reference*.

[3] Jochen Bomm, Andrea Hiller: *Anbindung und Retargeting von Motion Capture Daten in Maya*. Diplomarbeit, Hochschule der Medien, Stuttgart, Studiengang AM, 2002.

[4] Siegfried Bosch: *Algebra*. Springer-Verlag, Berlin, Heidelberg, New York, 2001.

[5] Armin Bruderlin, Lance Williams: Motion signal processing. *Computer Graphics Proceedings, Annual Conference Series*: 97–104, 1995.

[6] Bernhard Eberhardt: *Bewegungsaufnahme, -analyse und Bewegungsadaption*. Vortrag zur Habilitation, WSI/GRIS Universität Tübingen Sommersemester 2002.

[7] Bryan Ewert: *Maya API How-To*, 2000-2003.
`http://www.ewertb.com/maya/api/`

[8] R.Fisher, S.Perkins, A.Walker, E.Wolfart, 2000: *Spatial Filters - Mean Filter*.
`http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm`

[9] R.Fisher, S.Perkins, A.Walker, E.Wolfart, 1994: *Spatial Filters - Median Filter*. `http://www.dai.ed.ac.uk/HIPR2/median.htm`

[10] Michael Gleicher: *Motion Capture and Motion Editing: From Observation to Animation*. Vorab veröffentlichtes Kapitel 13: *Common Issues and Techniques - DRAFT*. AK Peters, 28. April 2001

[11] David A. D. Gould: *Complete Maya Programming*. Morgan Kaufmann, San Francisco, California, 2003.

[12] Florian Linner: *MotionLab - Bearbeitung von Motion-Capture-Daten mittels prozeduraler Verfahren*. Diplomarbeit, Hochschule der Medien, Stuttgart, Studiengang AM, 2000.

[13] M. Schafer: *Acclaim Skeleton File Definition*. Acclaim ATG, Glen Cove, NY, Februar 1995.
`http://www.motionrealityinc.com/software/asfamcspec.html`

[14] I. Stephenson: *A History of Live Action/Animation - Rotoscope*.
`http://www.geocities.com/SunsetStrip/Club/9199/Animation/Fleischer_Rotoscope.html`

[15] David J. Sturman: *A Brief History of Motion Capture for Computer Character Animation*.
`http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/motion_capture/history1.htm`

[16] Jan Svarovsky: *Quaternione für Spieleprogrammierung*. Kapitel aus dem Buch *Spieleprogrammierung Gems 1* mitp-Verlag, Bonn, 2002.

[17] Vicon: *Vicon Handbuch*.

[18] Webseite: `http://access.tucson.org/ michael/ejm_2.html`. Diese Seite dokumentiert die Entwicklung der Chronophotography

[19] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner: *OpenGL Programming Guide*. Addison Wesley, Reading, Massachusetts, Februar 2000.

Anhang A

Übersicht wichtiger Klassen

Hier werden die wichtigsten in *MotionTheater* verwendeten Klassen gezeigt. Es handelt sich dabei um die Klassen `MTMotionVector`, `MTMotion`, `MTImportASFFileCmd`, `MTImportAMCFileCmd` und `MTFilterCmd`. Die Auflistung enthält pro Klasse ebenfalls nur die wichtigsten Funktionen und Variablen.

MTMotionVector

```
class MTMotionVector    // repräsentiert ein Frame der Bewegung
{
    public:
        MTMotionVector();
        virtual ~MTMotionVector();

        MVector root_position;  // Position des Root-Knochens
        // Rotationsdaten aller Knochen
        map<string, MVector> joint_rotations_map;
        unsigned frame_number;  // Nummer dieses Frames
};
```

MTMotion

```
class MTMotion      // enthält alle Frames + Zusatzinformationen
{
    public:
        MTMotion();
        virtual ~MTMotion();

        // gibt Index zurück in Beziehung zu erstem Frame in File
```

```
        unsigned getIndex_file(unsigned frame_num);
        // diese Funktion gibt es auch noch für ersten "wirklichen"
        // Frame und ersten "wirklichen" Frame mit Blending

        // wahr, wenn Frame mit gegebenem Index in File existiert
        bool frameExists_file(int frame_index);
        // genauso gibt es hier Versionen für obige 2 Fälle

        vector<MTMotionVector> motion;      // die Bewegung selbst

        // Blendtiefen am Anfang und Ende
        unsigned blend_depth_start1, blend_depth_start2;
        unsigned blend_depth_end1, blend_depth_end2;

        // Framenummern, wo geblendet werden muss
        vector<unsigned> blendpres_frame_nums;

        // "wirkliche" Frames abhängig von cut & scale
        unsigned real_start_frame, real_end_frame;
        // Framenummern aus File
        unsigned file_start_frame, file_end_frame;
};
```

MTImportASFFileCmd

```
// Importiert ASF-File und erzeugt Skelett
class MTImportASFFileCmd : public MPxCommand
{
    public:
        MTImportASFFileCmd();
        virtual ~MTImportASFFileCmd();

        // Maya-spezifische Funktionen für MPxCommand
        virtual MStatus doIt(const MArgList& args);
        virtual MStatus undoIt();
        virtual MStatus redoIt();
        virtual bool isUndoable() const { return true; }
        static void *creator() { return new MTImportASFFileCmd; }

        // parst ASF-Header, Knochenhierarchie und erstes Frame
        MStatus importASFFile(const MString filename);
```

```
        // parst Zeilen des :hierarchy-Abschnitts und erstellt
        // Knochen entsprechend
        MStatus parseLine(C_Str line);
        // parst einen Knochen (Joint) aus :bonedata-Abschnitt
        MStatus parseNextJoint(bool &end_of_joints);

        // erzeugt Skelett (T-Pose) aus eingelesenen Daten in Maya
        MStatus setupCalibrationPose(void);
        // berechnet rekursiv Positionen aller Kind-Knochen
        MStatus evaluateChildren_pos(MObject parent_joint,
            const unsigned short child_index);


    private:
        MString filename_asf;           // ASF-Filename
        ifstream *asf_file;             // ASF-File

        map<string, MObject> joints;    // alle Joints zusammen bilden Skelett
        MVector root_pos;               // Rootposition im ersten Frame
        MEulerRotation root_orient;     // Rootorientierung im ersten Frame

        double scale_factor;            // globaler Skalierungsfaktor

        map<string, double> boneLengths;        // Knochenlängen aus ASF-File
        map<string, MVector> boneDirections;    // Knochenvektoren aus ASF-File
};
```

MTImportAMCFileCmd

```
// Importiert AMC-File = Bewegung
class MTImportAMCFileCmd : public MPxCommand
{
    public:
        MTImportAMCFileCmd();
        virtual ~MTImportAMCFileCmd();

        // Maya-spezifische Funktionen für MPxCommand
        // siehe class MTImportASFFileCmd

        // Hauptfunktion: extrahiert Bewegungsdaten aus AMC-File
        MStatus importAMCFile(const MString filename, unsigned start_frame,
```

```
        unsigned end_frame, MTMotion &motion_array, bool use_new_motion);

    // parst Daten aus allen Frames in AMC-File
    MStatus parseFrames(unsigned start_frame, unsigned end_frame,
        MTMotion &motion_array, bool use_new_motion);

    // erzeugt neue Animationskurven für gegebenen Knochen
    MStatus createAnimCurvesForJoint(string joint_name);
    // führt alle Vorbereitungen für Update durch
    MStatus prepareUpdate(void);
    // führt Update für gegebenen Knochen aus
    MStatus updateAnimCurves(string joint_name);

    // ermittelt Frames, an denen geblendet werden muss
    MStatus findBlendPreserveBorders(void);
    // überprüft und passt alle Blendtiefen an
    MStatus checkBlendDepths(void);

    // erzeugt motion_cutscaled aus motion_orig
    MStatus createMotionCutScaled(void);
    // erzeugt kombinierte Animationskurven für gegebenen Knochen
    MStatus combineMotions(string joint_name);


private:
    MString filename_amc;       // AMC-Filename
    ifstream *amc_file;         // AMC-File

    MTMotion *motion_orig, *motion_cutscaled, *motion_old;

    map<string, MObject> joints;    // alle Joints zusammen bilden Skelett

    // geparste Keyframewerte für Root-Position
    MDoubleArray xKeyValues_pos, yKeyValues_pos, zKeyValues_pos;
    // geparste Keyframewerte für Knochen-Rotationen
    map<string, MDoubleArray> xKeyValues_rot, yKeyValues_rot, zKeyValues_rot;

    // Variablen, die der Benutzer im UI angegeben hat
    unsigned cut_from, cut_to;      // cut
    unsigned scale_from, scale_to;  // scale
    int discard_old_motion;         // bei Update alte Bewegung wegwerfen?
```

```
        int blend_depth_user;           // Blendtiefe
        int preserve_posrot;            // alte Rootpos/rot erhalten

        double scale_factor;            // Skalierungsfaktor

        // Degrees of freedom für jeden Knochen, z.B. "xyz", "xz"...
        map<string, string> boneDofs;
};
```

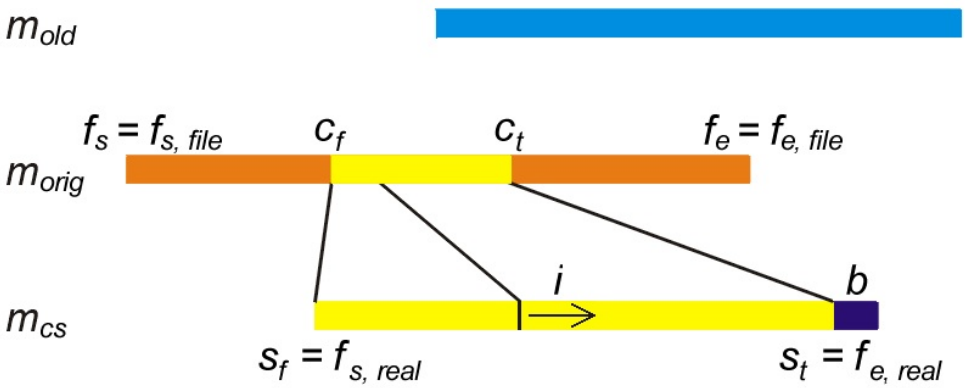
MTFilterCmd

```
// Filtert Animationskurven (3 verschiedene Filterarten)
class MTFilterCmd : public MPxCommand
{
    public:
        MTFilterCmd();
        virtual ~MTFilterCmd();

        // Maya-spezifische Funktionen für MPxCommand
        // siehe class MTImportASFFileCmd

        // durchläuft Bewegung & bereitet Filterung vor
        MStatus filterMotion(const unsigned short depth);

        // Filterfunktionen, die Filterung für 1 Wert durchführen
        void filterMedian(double &x_filtered, double &y_filtered,
            double &z_filtered);
        void filterMovingAverage(double &x_filtered, double &y_filtered,
            double &z_filtered);
        void filterWeightedAverage(double &x_filtered, double &y_filtered,
            double &z_filtered);

        // initialisiert Gewichte für gewichteten Mittelwertfilter
        void initWeights(void);

        // erzeugt neue Animationskurven aus gefilterten Werten
        MStatus updateAnimCurves(string joint_name, MDoubleArray &x_values,
            MDoubleArray &y_values, MDoubleArray &z_values);


    private:
```

```
        unsigned short filter_depth;    // Filtertiefe
        MString filter_type;            // Filtertyp

        // zu filternde Werte
        map<string, MDoubleArray> old_xValues, old_yValues, old_zValues;
        // gefilterte Werte
        vector<double> x_filterValues, y_filterValues, z_filterValues;

        // Filtergewichte für gewichteten Mittelwertfilter
        vector<double> filter_weights;

        // neue Animationskurven für Position & Rotation
        MFnAnimCurve animCurve_x, animCurve_y, animCurve_z;
        MFnAnimCurve animCurve_x_rot, animCurve_y_rot, animCurve_z_rot;
};
```

Anhang B

Grafische Darstellung der verwendeten Variablen



Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Stuttgart, den 01. Juni 2004